A Novel Approach to Asteroid Identification Using Image Processing of Existing Data

New Mexico Supercomputing Challenge
Final Report
March 31, 2009

Team 4
La Cueva High School

Team Members:
Erika DeBenedictis
Chris Hong
Tony Huang

Project Mentor:
Dr. Erik DeBenedictis

1. Table of Contents

2. Executive Summary

Asteroids are dangerous objects for Earth, as evidenced by previous asteroid impacts that have had planet-wide effects. It is therefore important to document the orbits of asteroids in our solar system to ensure fair warning should an asteroid approach Earth. Astronomical observatories worldwide constantly observe the sky to find new asteroids and update the orbits of known asteroids.

Computerized methods of analyzing longitudinal astronomical image data would greatly aid in this goal. Advances here could lead to the identification of previously unknown asteroids, greater understanding of the patterns in asteroid orbit perturbations, and more effective real-time astronomical monitoring of our night sky.

Our project focuses on this longitudinal image analysis. We have developed a conceptual framework for how existing astronomical image data may be analyzed for unknown object sightings as well as how these sightings may be further analyzed to document asteroids.

We have developed systems to find asteroids in images and accurately locate these images in the sky. In the process, we implemented enhancements to the methods we used to increase their accuracy. These enhancements are unique to our work.

We have begun to use the GPU (graphical processing unit) to analyze which observations of unidentified objects could be of the same asteroid. This process involves running orbit determination on a GPU, and is also original to our work.

3. Introduction

Asteroids are dangerous objects for Earth, as evidenced by previous asteroid impacts that have had planet-wide effects. Therefore, it is vitally important to carefully track the motion of all potentially dangerous asteroids in our solar system. This goal is difficult to achieve, in part, because the small mass of asteroids make their orbits easily perturbed, or changed, by the gravitational pull of other solar bodies. Further, not all asteroids have been discovered.

The primary mission of many well known observatories is to find new asteroids and update the orbits of previously known asteroids through continuous monitoring of the night sky.

A companion approach to finding and updating asteroid orbits is to analyze existing astronomical data rather than relying solely on new observations. A large-scale analysis of existing image data could lead to identifying new asteroids, to better understanding the patterns in asteroid orbit perturbations, and to more effective real-time observations. Our project focuses on this type of longitudinal image processing analysis.

3.1 Problem Statement

How can astronomical image data be automatically, accurately, and efficiently analyzed for possible asteroid observations? How can these observations be most effectively used to further our knowledge of objects in our solar system?

3.2 Procedure Overview

Our study begins with the acquisition of raw heterogeneous astronomical images for data analysis. Such images often come in standard astronomical data formats, which usually include an approximate location in the sky as part of the format. This location is used to query reference star catalogues to find what stars should be in the image.

Using this data, we then compare the stars in the image to the stars in the sky by finding similar triangles between corresponding stars. This determines the exact location and orientation of the image we are analyzing.

Any objects that are in the image then are translated to a location in the sky and recorded with the time of observation. This constitutes one object observation. These observations may be further analyzed to find previously unknown asteroids or patterns in asteroid perturbations.

One way to aid in finding asteroid observations that are of the same object involves calculating potential orbits for unknown object observations. We have implemented this compute-intensive process on the GPU, or Graphical Processing Unit, to aid in the discovery of new asteroids.

4. Background Research

4.1 Asteroid Perturbations

Since most asteroids have much smaller masses than planets and their moons, they are easily affected by the gravitational tugs of other celestial objects. These influences, while subtle, can completely alter an asteroid's orbital path. It is important for scientists and astronomers to continually update the orbits of asteroids, particularly the ones that are potential threats to the Earth. NEOs, or Near-Earth-Objects, need to be frequently monitored because a slight perturbation could send the asteroid directly toward the Earth.

Updating asteroid orbits is not a new area of science. In fact, there are multi-million dollar telescopes, such as the Magdalena Ridge Observatory, whose purpose is to track asteroids through the night sky. By refining an asteroid's orbit through careful observation, it is possible to increase the accuracy of its collision probability estimate with the Earth. An example of this is 99942 Apophis, an asteroid that was initially labeled with 2.7% chance of collision. However, after more accurate observations were made, this probability was lowered to a 1 in 45,000 chance (Brown). This shows the utility and importance of asteroid observations. In this case, the collision probability was reduced after orbit refinement based on detailed observations. What if the slight perturbations had increased, instead of decreased, the collision probability? It could be catastrophic to the planet if scientists were not aware of increased collision chances because they did not continuously monitor the orbits of NEOs.

4.2 Asteroid Identification

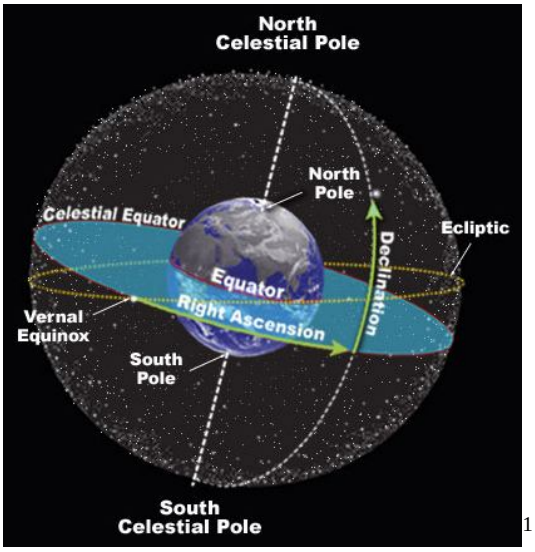
In an image of the night sky, an asteroid looks exactly like any star. Therefore, astronomers need methods to analyze an image and identify the asteroids. One can manually find asteroid observations by analyzing two images of the same region taken consecutively. By looking at any objects that have moved in between the images, it is possible to locate asteroids. However, this method is tedious and impractical because it requires two images that are identical except for the movement of the asteroid.

In our project we experiment with various automated methods to identify asteroids in images. These methods begin with identifying the centers of all the objects in the image, called centroids. The centroids reduce an image to a list of points, the basis for the rest of our asteroid identification process.

4.3 Equatorial Coordinate System

It is difficult to plot the night sky because of the constant motion of the Earth. Astronomers have developed a system known as Equatorial Coordinates to be able to pinpoint any location in the night sky regardless of the position and time on Earth. This coordinate system assumes that the Earth is in the center and all the celestial objects are in a sphere surrounding the Earth. The plane of the Earth's axes is offset approximately 23.5 degrees from the ecliptic, or the

plane of the sun.



[1]

The two units in this coordinate system are Right Ascension (RA) and Declination (Dec). Right Ascension is similar to longitude on Earth, and is parallel to the Earth's axes. Zero degrees RA is defined as the point on the celestial sphere where the ecliptic plane goes through the equatorial plane and RA increase counter-clockwise from there. Declination is similar to latitude lines on Earth, and is perpendicular to the axes of Earth. Zero degrees Dec is the celestial equator and increases as we approach the North Celestial Pole. Using this system, astronomers can give a unique set of coordinates to every star in the sky.

4.4 Locating a Picture in the Sky

In order to find the location of a picture in the sky, it is necessary to devise a method of determining the exact celestial coordinates of each star in the image. Traditionally this is done, at least partially, by hand. Researchers select several stars in the image and then, using the celestial coordinates of these reference stars as well as their pixel coordinates, determine a translation from (x,y) to (Ra, Dec). These celestial coordinates pinpoint any location in the image.

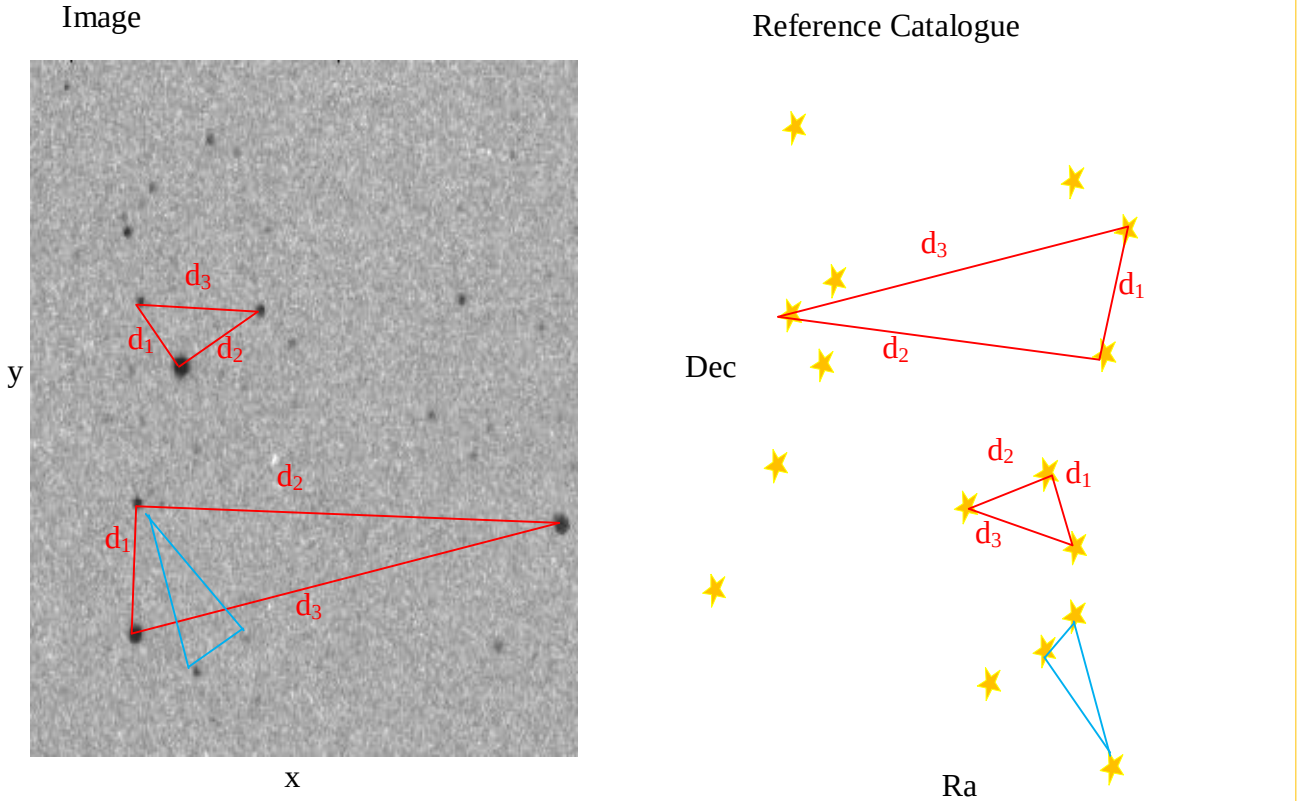
Padgett, Kreutz-Delgado, and Udomkesmalee's paper "Evaluation of Star Identification Techniques" details three methods to automatically find an image's orientation and position in the sky. The methods they examine are often used as backup for satellite guidance systems, and therefore database size and required compute power are primary concerns. The paper compares three methods that reduce the image into a searchable pattern by varying degrees.

The method they found to be most effective for satellite guidance was called the 'grid method,' which turns each image into a series of very low-resolution black and white images, usually 20 by 20 pixel 'grids,' aligning each with different pairs of stars on a fixed axis. These grids may then be easily searched against a database of similarly formed catalogue patterns.

[1] http://www.astro.virginia.edu/class/whittle/astr130/im/RA-Dec-celsphere.jpg

Although they found the grid method to be the most efficient for their goal, they did not implement some of the optimizations we took into account when using another method they called the 'triangle method.' They also assume that the spacecraft has no prior knowledge of where it pointed, whereas for our application we have an approximate location in the sky for the images we wish to align. For their purpose, it is especially important to minimize the time needed to search an extensive database. Our application increases the importance of minimizing the time needed to create the search elements.

For these reasons, we chose to use the triangle method as the best method to locate our image in the sky. In this method we have a list of centroids of stars in the image in (x,y) and a list of stars in the approximate area of the sky in (Ra, Dec). We draw triangles between the centroids and the stars, and then search for similar triangles by comparing the ratio of side lengths. Triangles between corresponding stars in a set of points will form similar triangles. From the coordinates of the vertexes of these triangles, we can calculate scale, rotation, and translation between (x,y) coordinates and (Ra, Dec).



The triangle method finds congruent triangles between two sets of points, in this case the centroids in an image and the corresponding reference star catalogue. Some triangles, the red triangles in this case, have closely corresponding side length proportions and are used to find scale, rotation, and translation between the two images. Others, the blue triangles, are false close matches.

We cannot be sure that we have chosen the same stars to form triangles in the image and in the sky, so there will be many triangles that have no correct match. There also will be triangles

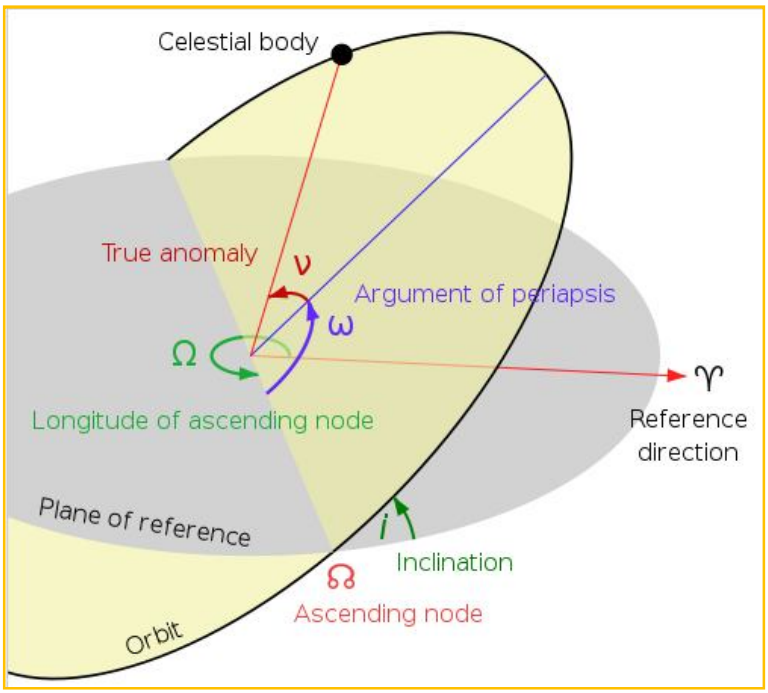
between non-corresponding stars that happen to be similar. These triangles must be automatically eliminated before the true scale, rotation, and translation may be calculated through various filtering techniques to ensure an accurate conversion between (x,y) and (Ra, Dec).

4.5 Analysis of Asteroid Observations

Once a list of asteroid observations is made, it is possible to determine if any of the observations belong to the same asteroid. To do this, it is necessary to analyze the orbits created by the asteroid observations. Six unique orbital elements define a unique elliptical orbit for an asteroid around the sun. These orbital elements are as follows:

The first two orbital elements are *a* (semi-major axis) and *e* (eccentricity). These elements define the ellipse that the asteroid orbits in.

The next two orbital elements are *i* (inclination angle) and Ω (longitude of ascending node). These elements define the orientation of the asteroid's orbital plane with the ecliptic plane, or the plane of the sun.



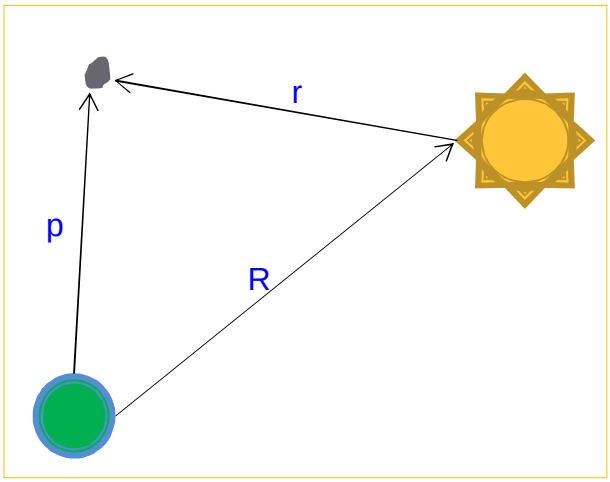
As shown in the diagram, *i* is the angle between the orbital and the ecliptic plane. Setting *i* will lock the position of the orbital plane in one aspect. However, the orbital plane can still rotate horizontally, so another orbital element is needed. The longitude of ascending node Ω is defined as the angle between the reference direction (x-axis) and the line of ascending node. This line is where the orbital plane ascends through the ecliptic plane. The combination of *i* and Ω lock the orbital plane's orientation with the ecliptic plane.

The fifth orbital element is the ω (argument of perihelion). Even though the orbital plane is now locked into position, the orbit of the asteroid can still rotate within the orbital plane. The argument of perihelion is the angle between the ascending node and the perihelion of the orbit, or

the point in the orbit at which it is farthest from the sun.

The last orbital element is *M* (mean anomaly). This orbital element describes where the asteroid is along its orbit, so is the only element that changes with time. The orbital elements combine to determine a unique orbit for an asteroid.

In order to determine the values of the orbital elements, at least three observations of the same asteroid are needed. Three observations will define a unique set of the orbital elements. Once the RA and Dec for three asteroid observations are recorded, then it is possible to solve the Fundamental Vector Triangle Problem using the Method of Gauss. The Method of Gauss is essentially a two-body problem taking into account the gravitational attractions of the Earth and the Sun. The Earth is only an observational body in the problem, and its gravitational pulls are not calculated.



In this method, there are three vectors. Vector p is the Earth-Asteroid vector. This is the vector that is observed and calculated from the image analysis process. R is the Earth-Sun vector, which is known and well documented. By subtracting these two vectors, it is possible to calculate r, or the orbital vector. The actual process of solving this vector problem is very complicated, and we have provided a brief summary below. These equations are exactly implemented in the code, this is why many of them simply re-define variables for readability.

1. We input nine numbers into the problem. The method requires three observations, each with a Julian date, RA (α), and Dec (δ).
2. Based on the RA and Dec for each observation, we can create a unit-vector L that is equivalent to $\frac{\vec{p}}{p}$, or the unit vector pointing from the Earth to the asteroid. Then, through coordinate transforms, we can find vector L to be equal to

$$\overrightarrow{L_i} = \langle \cos\delta_i \cos\alpha_i \, , \cos\delta_i \sin\alpha_i \, , \sin\delta_i \rangle$$

for i=1, 2, 3 for our three observations.
3. We can calculate vector R for each observation. This is done by generating an ephemeris for the sun with respect to the Earth. A description of the ephemeris generator

we used can be found in the Code Overview.

4. We need to define the times of our observations relative to each other. This is done though calculating proper time, or τ. In these equations, k is the Gaussian gravitational constant. We also define $\Delta\tau$ as the difference between τ_3 and τ_1.

$$\tau_1 = k(t_1 - t_2)$$
$$\tau_2 = 0$$
$$\tau_3 = k(t_3 - t_2)$$

5. Next, we calculate the D-coefficients, which are as follows. These values will eventually allow us to determine p, or the distance between the Earth and the asteroid.

$$D_0 = \overrightarrow{L_3} \cdot (\overrightarrow{L_1} \times \overrightarrow{L_2})$$
$$D_{1j} = \overrightarrow{L_3} \cdot (\overrightarrow{R_j} \times \overrightarrow{L_2})$$
$$D_{2j} = \overrightarrow{L_3} \cdot (\overrightarrow{L_1} \times \overrightarrow{R_j})$$
$$D_{3j} = \overrightarrow{L_1} \cdot (\overrightarrow{L_2} \times \overrightarrow{R_j})$$

6. We also assign A and B coefficients for the quantities based on time. Once again, these coefficients are shorthand for the eventual calculation of p.

$$A_1 = \frac{\tau_3}{\Delta\tau} \qquad A_3 = -\frac{\tau_1}{\Delta\tau}$$
$$B_1 = \frac{A_1}{6}(\Delta\tau^2 - {\tau_3}^2) \qquad B_3 = \frac{A_3}{6}(\Delta\tau^2 - {\tau_1}^2)$$

7. We then define the A, B, E, and F coefficients. This is purely used to get the coefficients that will allow us to solve an eighth-degree polynomial. This will eventually give us an initial estimate for the asteroid's distance from the sun.

$$A = -\frac{A_1 D_{21} - D_{22} + A_3 D_{23}}{D_0}$$
$$B = -\frac{B_1 D_{21} + B_3 D_{23}}{D_0}$$
$$E = -2(\overrightarrow{L_2} \cdot \overrightarrow{R_2})$$
$$F = {R_2}^2$$

8. We then find the a, b, and c coefficients, where μ is equal to 1.

$$a = -A^2 + AE + F$$
$$b = -\mu(2AB + BE)$$
$$c = -\mu^2 B^2$$

9. Using Newton's Method, we can solve for r_2. This gives us the distance from the Sun to the asteroid at the time of the second observation, or the middle observation. This is important because it gives us an initial approximation that we can improve to solve the orbital vector.

$${r_2}^8 + a{r_2}^6 + b{r_2}^3 + c = 0$$

10. Use r_2 to approximate f and g series. These are Taylor polynomials that will allow us to estimate the r vector, or the orbital vector. For now, we will only take two

iterations, but we will refine this later in the process. For i = 1,3

$$f_i = 1 - \frac{1}{2}u{\tau_i}^2$$

$$g_1 = \tau_i - \frac{1}{6}u_2{\tau_i}^3$$

$$u_2 = \frac{\mu}{{r_2}^3}$$

11. Now we get the c-coefficients. We will use this in the determination of the p vectors.

$$C_1 = \frac{g_3}{f_1 g_3 - f_3 g_1}$$

$$C_2 = -1$$

$$C_3 = \frac{g_1}{f_1 g_3 - f_3 g_1}$$

12. As well as the p-coefficients. These are the distance between the Earth and the asteroid at all three observations.

$$P_1 = \frac{C_1 D_{11} + C_2 D_{12} + C_3 D_{13}}{C_1 D_0}$$

$$P_2 = \frac{C_1 D_{21} + C_2 D_{22} + C_3 D_{23}}{C_2 D_0}$$

$$P_3 = \frac{C_1 D_{31} + C_2 D_{32} + C_3 D_{33}}{C_3 D_0}$$

13. Now we can finally find the orbital vectors r.

$$\vec{r_i} = P_i\vec{L_i} - \vec{R_i}$$

14. We next evaluate the d-coefficients. Once again, these are primarily used to clean up the equations for further use.

$$d_1 = \frac{f_3}{f_3 g_1 - f_1 g_3}$$

$$d_3 = -\frac{f_1}{f_3 g_1 - f_1 g_3}$$

15. We then find the derivative of the r-vector to find the orbital velocity vector. This involves the use of the orbital vectors from the other two observations.

$$\dot{\vec{r_2}} = d_1\vec{r_1} + d_3\vec{r_3}$$

16. We then perform several steps to refine the orbital velocity.

    16.1 – First, we apply the correction for light travel time.

$$t_{ci} = t_i - \frac{P_i}{173.1446}$$

$$\tau_1 = k(t_{c1} - t_{c2})$$

$$\tau_3 = k(t_{c3} - t_{c2})$$

$$\Delta\tau = \tau_3 - \tau_1$$

    16.2 - Then, we re-evaluate the f and g series, keeping up to 4th order terms this time.

This will give us more accuracy in this process.

$$f_i = 1 - \frac{u_2}{2}{\tau_1}^2 + \frac{u_2\xi_2}{2}{\tau_i}^3 + \frac{1}{24}\left(3u_2Q_2 - 15u_2{\xi_2}^2 + {u_2}^2\right){\tau_i}^4 + \cdots$$

$$g_i = \tau_1 - \frac{u_2}{6}{\tau_1}^3 + \frac{u_2}{4}\xi_2{\tau_i}^4 + \cdots$$

Where $u_2 = \frac{\mu}{r_2}$, $\xi_2 = \frac{\vec{r_2}\cdot\dot{\vec{r_2}}}{2}$, and $Q_2 = \frac{\vec{r_2}\cdot\dot{\vec{r_2}}}{r_2}$

16.3- We then perform steps 11-15 again.

16.4 - Finally, we use the new orbital velocity to redo this loop again. This will eventually gives us accurate orbital vectors.

17. At this point, we have refined the orbital vector. The next steps are to extract the orbital elements of the asteroid from this orbital vector and velocity. First, we convert to elliptical coordinates from equatorial. This is because the asteroid orbits the Sun, not Earth, and we should base our coordinates with the Sun at the center. In this equation, epsilon is approximately 23.45 degrees, or the angle between the plane of the sun and the Earth.

$$\overrightarrow{r_2}^{EC} = x_2\hat{x} + \left(y_x\cos\varepsilon + z_2\sin\varepsilon\right)\hat{y} + \left(-y_2\sin\varepsilon + z_2\cos\varepsilon\right)\hat{z}$$

$$\dot{\overrightarrow{r_2}}^{EC} = \dot{x_2}\hat{x} + \left(\dot{y_x}\cos\varepsilon + \dot{z_2}\sin\varepsilon\right)\hat{y} + \left(-\dot{y_2}\sin\varepsilon + \dot{z_2}\cos\varepsilon\right)\hat{z}$$

18. We define an h-vector. This is a constant vector for a particular asteroid because from Kepler's 2nd Law, an orbiting body will "sweep" the same amount of area in its orbit over a constant amount of time no matter where it is in its orbit. This will cause asteroids to move slower when farther away, and faster when closer to the sun.

$$\vec{h} = \overrightarrow{{r_2}^{EC}} \times \dot{\overrightarrow{{r_2}^{EC}}}$$

19. We find the first orbital element *a*, or the semi-major axis. This is derived from the relationship between orbital speed and position, as given by Kepler's 2nd Law.

$$a = \left(\frac{2}{{r_2}^{EC}} - \frac{\dot{\overrightarrow{{r_2}^{EC}}}\cdot\dot{\overrightarrow{{r_2}^{EC}}}}{\mu}\right)^{-1}$$

20. The next element is *e*, or eccentricity.

$$e = \sqrt{1 - \frac{h^2}{\mu a}}$$

21. The third orbital element *i*, or inclination can be calculated from its cosine value given that it is $0 \le i \le 180$. Vector h is always perpendicular to the orbital plane, and this allows us to find the angle between the orbital and ecliptic plane.

$$\cos i = \frac{h_z}{h}$$

22. The fourth orbital element Ω, or the longitude of ascending node can be calculated from the known values of both its sine and cosine.

$$\cos\Omega = \frac{h_z}{h\sin i} \qquad \sin\Omega = \frac{h_x}{h\sin i}$$

23. We then calculate U, also from its known cosine and sine values. U is time dependent, but we can use it to find our fifth orbital element.

$$\cos U = \frac{{r_2}_x^{EC}\cos\Omega + {r_2}_x^{EC}\sin\Omega}{r_2^{EC}} \qquad \sin U = \frac{{r_2}_z^{EC}}{r_2^{EC}\sin i}$$

24. We then calculate V, also from its known cosine and sine values. Like U, it is also related to time, but it changes at the same rate as U. Therefore, by subtracting V from U, then it is possible to define a constant value.

$$\cos V = \frac{1}{e}\left(\frac{a(1-e^2)}{r_2^{EC}} - 1\right) \qquad \sin V = \frac{a(1-e^2)}{e}\left(\frac{\dot{\overrightarrow{r_2^{EC}}}\cdot\overrightarrow{r_2^{EC}}}{hr_2^{EC}}\right)$$

25. The fifth orbital element ω, or the angle of perihelion can be calculated by ω=U-V.
26. Next, we calculate E, which will always lie in the same quadrant as V. E is the eccentric anomaly, and gives a relation between geometry and time for the orbit.

$$\cos E = \frac{1}{e}\left(1 - \frac{r_2^{EC}}{a}\right)$$

27. Finally, we get the last orbital element M, by $M = E - esinE$. This is the mean anomaly, and defines the location of an asteroid in its orbit.

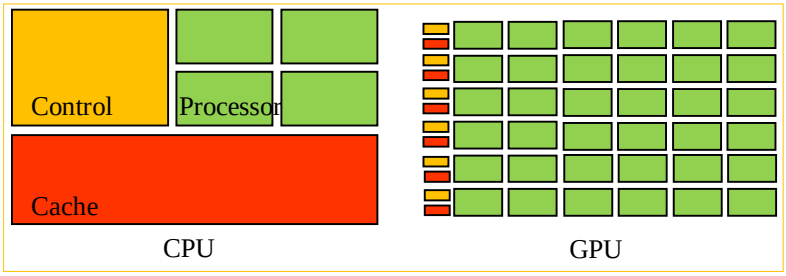
From the list of all the possible asteroid sightings, we run an orbital determination process for every subset of three observations. From here, it is possible to determine which of the orbits are reasonable. For example, if the generated orbit has a semi-major axis of 10000 AU's (astronomical units), then the orbit is most likely wrong because the majority of asteroids lie within the solar system. Furthermore, if an orbital element turns out negative, then the orbit is also incorrect.

Once we determine which asteroids appear in our images, we can begin the process of checking for perturbations. Using an ephemeris generator based on our orbital elements, we can determine the position of an asteroid at any time. Therefore, we can obtain images of the regions where an asteroid is scheduled to be at a certain time and check for asteroids in the image. If we find that an asteroid is no longer where it should be according to our initial orbit, then we can conclude that its orbit has been perturbed. It is then possible to backtrack the asteroid's orbit to the area where it is first perturbed. The more observations of an asteroid there are, the more refined its orbit can be. We believe this is extremely important in our goal to analyze perturbations and update asteroid orbits. An easier method to cite and identify asteroids is one of our main motivations in this project.

4.6 The GPU Architecture

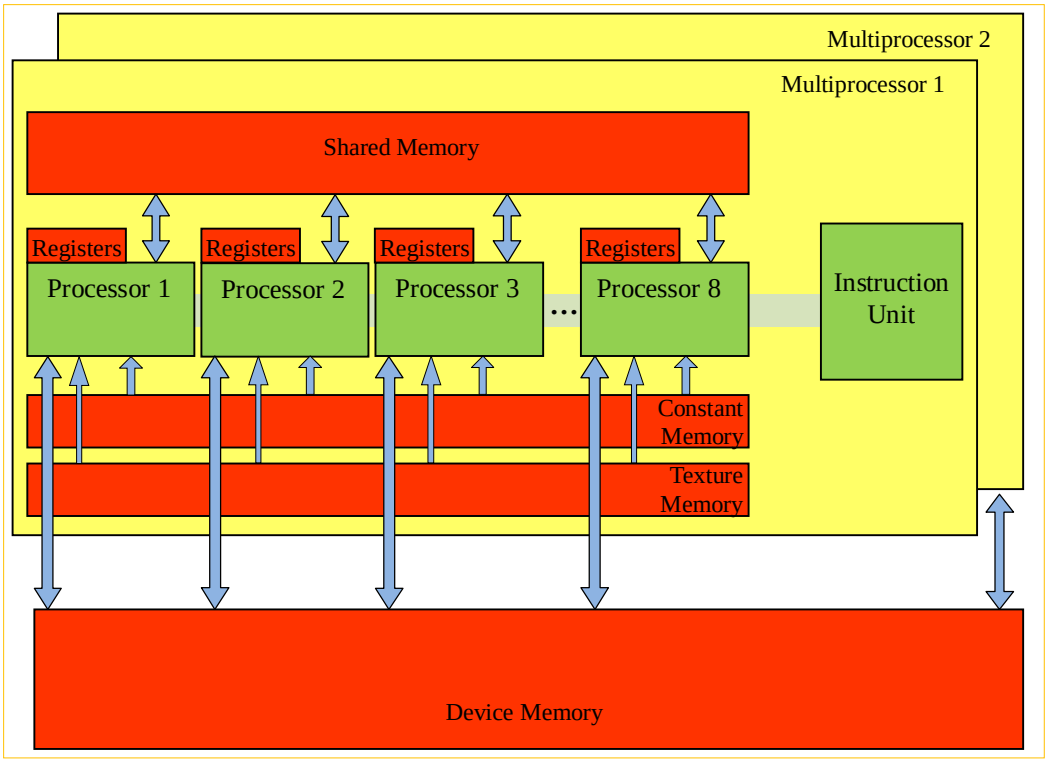
Graphical Processing Units, or GPUs, are specifically designed to meet compute-intensive graphics applications such as rendering. In recent years, methods of programming GPUs for applications other than graphics have become available, most notably NVIDIA's CUDA compiler. These advances have become known as General Purpose GPU (GPGPU), where the compute power of the GPU is used for other massively parallel applications beyond graphics.

The structure of a GPU can be compared to a multi-core processor but with many more processors and less on-board memory, as shown below.[2] CPUs have large amounts of control over how computations are scheduled for processors as well as relatively large amounts of fast cache memory. This allows them to perform versatile tasks. The GPU has far more processors but far less control over what these processors compute. In fact, each processor on a GPU must execute the same calculation at any given time. Also, computations that are conditional are handled differently on the GPU; if-statements, for example, are always scheduled for computation regardless of whether or not the conditional was true because the GPU has so little hardware devoted to controlling processes.



The GPU's hardware configuration makes it perfect for graphics processing, where the same compute-intensive calculations are performed on very little data. Another aspect of the GPU's graphics origins is that it does not support highly accurate calculations because it uses single-precision. The low-memory requirements of graphics also make the amount of cache limited. Below is a more detailed image of the types of memory on a GPU.

[2] GPU architecture pictures similar to those in the NVIDIA CUDA Programming Guide.



The GPU used in this project NVIDIA GT 8600, which has two multiprocessors with eight processors each. Other GPUs have different numbers of processors and multiprocessors. Each processor has immediate access to its registers. The shared memory for each multiprocessor is read-write and constant and texture memory is read-only for all processors. Loading data from the CPU and interface between the multiprocessors is achieved through device memory, which has an overhead of over 100 times all the other types of memory, and is used as little as possible.

The GPU's huge amounts of compute power are difficult to exploit in other applications because of its specialized configuration. Applications other than graphics that will run well on the GPU are those that require little memory, are massively parallel, have few conditional statements, and have large amounts of computation. There are few such applications.

In our project we use the GPU to determine if object sightings could potentially be of the same asteroid. This process is detailed in section 6.4.

5. Procedure: Image Analysis

Image processing consists of an image input, such as a photograph; the output can be either an image or a set of characteristics related to the image. In our project, image processing has allowed for the extraction of specific, valuable information from pictures.

5.1 Sources of Image Data

This project focuses on the analysis of heterogeneous astronomical images. For this reason, we want to refine our software approach with data from a variety of telescopes in different locations worldwide with varying filters and recorded information. The goal is to create robust analysis methods that can work for this wide variety of images.

Many observatories use similar methods as those employed by this project to extract the relevant data out of memory-intensive images. Hawaii's Pan-STARRS Observatory, for example, collects around 10 TB of image data a night, which is processed and reduced down to relevant measurements, disregarding the original image data (Scientific Detectors for Astronomy). Even for smaller projects, astronomical data is usually proprietary. For these reason, data is not readily available.

We used several sources of data for this project. Mr. Bill Wallace, an Albuquerque Astronomical Society member, provided us with some of his images. We also used a publicly available catalog of original pictures from the ESO (European Southern Observatory) organization, which provides original pictures from a variety of telescopes in the Southern Hemisphere. Mr. Holmes of The Astronomical Research Institute provided us access to even more images. These pictures are downloaded in zipped form through their FTP website. The pictures themselves are in a standard astronomical format called FITS files, which is a versatile format, containing several options of data types that are common in astronomy. They all contain a header file, in which telescopes automatically record the Julian date, Ra, Dec, latitude, longitude of each image. To read these files, we used the CCfits library to extract the relevant data and convert the image into a format we could read more easily. The FITS file format allows images to be easily shared between astronomers because of the completeness of their data.

Although file formats have been designed to be complete, archives such as the ESO plate library are rare because of the amount of memory required to hold original images. It is not standard practice to store raw data. Most publicly accessible databases contain reduced data, such as information about the variability of stars and their spectra, rather than images. Some sources do contain images to help astronomers recognize stars visually, but these images are usually conglomerations of surveys of the sky over many nights, and so are not time dated. While these types of images are not suitable for our purpose because we need to exact time of an observation, they can be immensely useful when constructing test data. For example, we used the USNO Image and Catalog Archive Server extensively to construct test data, as it returns actual images that are not precisely time dated.

5.2 Reference Stars

While there are relatively few sources of original images of the sky, star catalogs are relatively plentiful. The source we chose is the VizieR Service, an online system that allows access to hundreds of different catalogs. Searching this database can be done by constructing a hyperlink containing the desired catalogs, search window, and position, allowing the computer to automatically access this online information. We have selected several different catalogs for different purposes. Bright stars in an image are found using the UCAC2 Bright Star Supplement catalog and the All-sky Compiled Catalogue of 2.5 million star. A more general catalog of all stars in an image is found using The UCAC2 Catalogue.

VizieR Result Page

Result of VizieR Search within **2'** of **0. 0** (no other constraint specified)

Modify the Query

Max. Entries: 300 | Output layout: ;-Separated-Values | **ALL** columns | ReSubmit

I/289/out

UCAC2 Catalogue (Zacharias+ 2003)
The Second U.S. Naval Observatory CCD Astrograph Catalog *(48330571 rows)*

To get all details for a row, just click on the row number in the leftmost 'Full' column.

Note: *See also the UCAC home page at http://ad.usno.navy.mil/ucac/*

Full	2UCAC	RA(ICRS) deg	DE(ICRS) deg	e mas	e mas	UCmag mag	No	Nc	pmRA mas/yr	pmDE mas/yr	2Mkey	Jmag mag	Kmag mag
1	31794329	359.8817062	-00.0596075	20	15	14.36	3	2	4.8	-7.7	1178710943	12.977	12.397
2	31794332	359.9160483	-00.0028892	15	27	14.66	4	2	-5.1	-18.6	1178711025	13.637	13.326
3	31794333	359.9167683	-00.0519303	15	21	11.85	4	3	3.9	-15.6	1178710957	10.490	9.972
4	31794346	359.9990262	-00.0653548	13	18	10.02	4	6	8.3	-9.8	224910131	8.051	7.232
5	31613518	000.0107595	-00.0727045	15	29	13.10	4	2	20.5	-5.0	224910112	12.159	11.792
6	31613524	000.0306109	-00.0062920	47	15	15.78	2	2	10.5	-10.4	563750594	14.430	13.923
7	31613526	000.0374318	-00.0648148	13	18	10.51	4	5	-55.5	-90.5	563750476	9.478	9.077
8	31613529	000.0473686	-00.0067025	15	63	14.64	3	2	0.2	-9.0	563750590	13.514	13.167
9	31613531	000.0555612	-00.0787267	15	27	13.79	4	2	0.9	-9.7	563750437	12.658	12.328
10	31967478	359.8883192	+00.0541925	15	21	12.18	4	3	-22.7	-16.5	1098561822	11.311	10.965
11	31967481	359.8978050	+00.0012303	42	38	15.33	3	2	27.6	-2.0	1178711034	13.991	13.461
12	31967487	359.9895868	+00.0122900	45	44	15.31	3	2	2.3	-17.6	224910248	13.750	13.140
13	31794351	000.0390103	+00.0015342	53	52	15.46	3	2	-14.9	14.2	563750616	12.682	11.844
14	31794359	000.0745021	+00.0098956	15	61	15.92	2	2	1.5	-13.7	563750632	14.727	14.038
15	31794365	000.1208709	+00.0005300	15	15	14.14	5	2	-10.4	-4.6	563750613	12.180	11.369

We have developed a program to automatically query these catalogs, download the HTML page they return, and allow it to be used in the rest of our calculations. Above is a sample of a page VizieR might return, from which our program parses and extracts the needed information.

5.3 Centroids

Used in image processing, image moments are certain particular weighted averages of the image pixels' intensities or functions of those moments. The central moment, or centroid, of an image is defined (Shutler) as

$$\mu = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty}(x-\bar{x})(y-\bar{y})f(x,y)\,dx\,dy$$

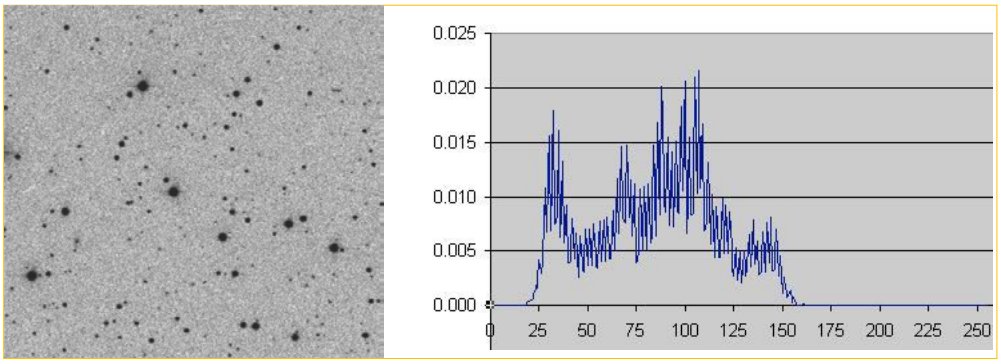
Where $\bar{x} = \frac{M_{10}}{M_{00}}$ and $\bar{y} = \frac{M_{01}}{M_{00}}$ are the components of the centroid.

If f(x,y) is a digital image, then the previous equation becomes

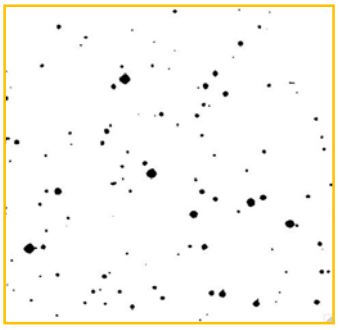
$$\mu = \sum_x \sum_y (x - \bar{x})(y - \bar{y}) f(x, y)$$

As it pertains to our project, the first segment of image processing consists of preparing the image for and the performing of centroid computation. This is a necessary step because the observation images we are given have heavy background noise and each star is irregularly shaped and varies in pixel size. By calculating the centroid for each star, we obtain its accurate, x-y coordinate location.

In preparing the image, background noise must be removed. This is attained through thresholding, a technique that converts the grayscale image to a binary image based upon a threshold value. If a pixel in the image has an intensity value less than the threshold value, the corresponding pixel in the resultant image is set to black. Otherwise, if the pixel intensity value is greater than or equal to the threshold intensity, the resulting pixel is set to white. As for choosing a reasonable threshold value, we used a histogram. The histogram consists of the vertical axis being a ratio between a pixel intensity value of "x" to the total number of pixels in an image. The horizontal axis will contain the pixel value "x". A sample grayscale image and its corresponding histogram are seen below.



Looking at the picture on the left, we can see that there are no white pixel intensity values. From the histogram representation on the right, we can see that the most dominant pixel value is 107. So if we select a threshold value of 107 and make every pixel with an intensity below the threshold black, and every pixel with an intensity above the threshold white, we come up with the following binary image:



To further clean the image, a blurring reduction technique is used in which a 9 by 9 window is created around each pixel in the binary image. Then, the median value of these 9 pixels is taken. If the median is 0, then the center pixel is set to black. If the median is 1, the center pixel is set to white (remember the image is inverted, so stars are black).The resulting stars are much more defined and clearly separated.

The last step of the preparation phase is grouping and labeling each star's region based on pixel connectivity. The process works by scanning an image, pixel-by-pixel (from top to bottom and left to right) in order to identify connected pixel regions, i.e. regions of adjacent pixels which share the same set of intensity values V. (For a binary image V={0}). The labeling operator scans the image by moving along a row until it comes to a point p (where p denotes the pixel to be labeled at any stage in the scanning process) for which V={0}. When this is true, it examines the four neighbors of p which have already been encountered in the scan (i.e. the neighbors (i) to the left of p, (ii) above it, and (iii and iv) the two upper diagonal terms). Based on this information, the labeling of p occurs as follows:

a. If all four neighbors are 1, assign a new label to p, else

b. If only one neighbor has V={0}, assign its label to p, else

c. If more than one of the neighbors have V={0}, assign one of the labels to p and make a note of the equivalences.

After completing the scan, the equivalent label pairs are sorted into equivalence classes and a second label is assigned to each class. As a final step, a second scan is made through the image, during which each previous label is replaced by the second label assigned to its equivalence class. For display, the final labels are shown by their different gray levels.

Now that every star has its own label, it becomes fairly simple to step through the image and record the pixel coordinates and brightness value for every pixel in each star. Now that all the necessary information has been found, we can perform a centroid calculation, which is essentially calculating the x-y pixel center of each star weighted by brightness.

$$X_{weighted} = \frac{\sum_i (X_i * Brightness_i)}{\sum_i Brightness_i} \quad Y_{weighted} = \frac{\sum_i (Y_i * Brightness_i)}{\sum_i Brightness_i}$$

This allows us to reduce the image down to the coordinates of the center of stars and objects.

5.4 Triangle Method

Now that we know the pixel coordinates of objects in our image from centroid calculation as well as the celestial coordinates of the reference stars in the field, we can use the triangle method to find a conversion factor between (x,y) and (Ra, Dec). In order to make this process efficient and effective in finding an accurate result, we modified the process for optimal

performance, as discussed in our "Analysis of Methods Section."

The resulting conversion factor allows us to immediately know the celestial coordinates of any objects we find in the image. Combined with the Julian date on which the photo was taken, we can record these possible asteroid observations.

The triangle method is also used between multiple images of the same star field to aid in the identification of possible asteroids in the image.

5.5 Identifying and Recording Possible Asteroids in Images

Once both images have been aligned by rotation, scale, and translation, we can find a conversion factor between the two images and transcribe the centroids of one image into the sky or into another image.

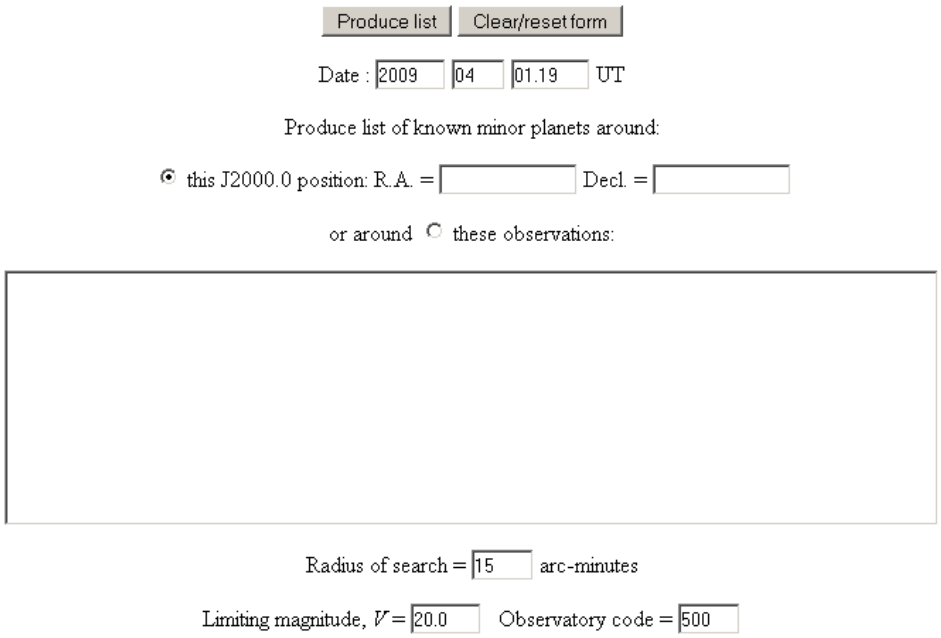
There are several methods of identifying asteroids in images. The first is to use two pictures of the same area. Any centroid in one image whose area does not lie within a centroid's area in the other image is considered to be potential asteroid. However, given images with varying exposure, often dim stars on the threshold of being filtered out appear in one image but not the other, creating false positives for asteroid observations.

Another method is to compare the centroids in an image to a list of all the stars in the image's area. This method still has the possibility of creating false positives because often star catalogs do not include every star in a range of brightnesses. A combination of the two methods, finding possible new asteroids between two images and then verifying that they are not dim stars, was used in this project.

Once possible asteroids observations have been found in an image, they are recorded with information pertaining to the name of the image they were found in, their celestial coordinates, and the precise time of observation taken from the FITS header file.

5.6 Determining Which Asteroid is in an Observation and Perturbations

It is possible to identify an asteroid solely by its celestial coordinates at a precise time. Through our image analysis, we obtain a set of celestial coordinates for any possible asteroids in an image. From error analysis, we are confident that these coordinates are accurate, and can be paired with the time at which the image was taken. With these two pieces of information, we can use the MP Checker (Minor Planet Checker http://scully.cfa.harvard.edu/~cgi/CheckMP) to identify our asteroid. In this online database, we can designate an area of the sky (usually around 5-15 arcminutes across), and check for the presence of any asteroids at any time.



The database will output a list of all known asteroids that will appear in this window at our designated time.

The following objects, brighter than $V = 20.0$, were found in the 10.0-arcminute region around R.A. = 01 01 01, Decl. = +05 29 05 (J2000.0) on 2009 04 01.19 UT:

Object designation	R.A. (h m s)	Decl. (° ' ")	V	Offsets R.A.	Offsets Decl.	Motion/hr R.A.	Motion/hr Decl.	Orbit	Further observations? / Comment (Elong/Decl/V at date 1)
(1615) Bardwell	01 01 25.8	+05 36 19	16.5	6.2E	7.2N	63+	25+	28o	None needed at this time.
(28168) 1998 VY25	01 01 00.9	+05 38 51	19.5	0.0W	9.8N	66+	29+	9o	None needed at this time.

Since the database allows searches for windows up to 900 arcminutes, we are confident that we can identify any asteroids found in our images since the error from image analysis is dwarfed by the search size. In the case that no asteroid is found in the region, it may be possible that the asteroid found from image analysis is unknown. Another possible cause for the lack of matching may be perturbations.

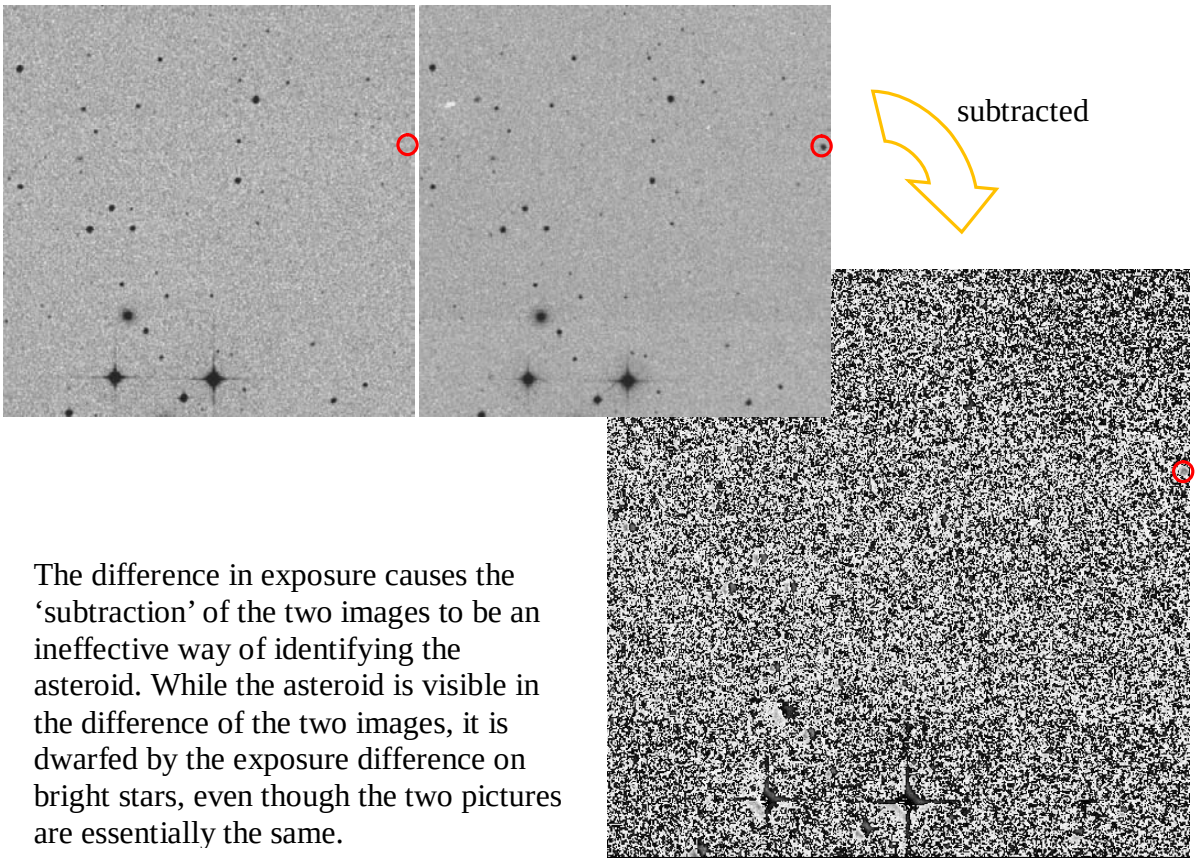
From our accurate image analysis results, it is possible to detect asteroid perturbations. Using the Minor Planet Checker, we can find all asteroids that are supposedly in a particular region at any time. Then, we can obtain images of that region at the exact time, and perform our image analysis. Since the error of the image analysis is a few pixels at most, we can be confident in our conclusion that the asteroid has been perturbed if it does not appear in the image. Of course, we need to confirm that certain conditions are met first, such as ensuring that the asteroid is bright enough to be caught by the telescope. To confirm a perturbation, we may need to perform this image analysis for several different celestial locations and times.

6. Analysis of Methods

Analyzing diverse data is challenging because it requires adaptable methods that work for a wide variety of images. During the course of our research, we have discovered many methods that make the baseline idea more applicable to diverse data, more accurate, or more efficient, as presented in this section.

6.1 Methods of Identifying Asteroids in Images

We had to rework the actual method to find asteroids to account for exposure differences between different images. Our original idea to compare the centroids from two images was to take the Fast Fourier Transform (FFT) of each, subtract the two frequency representations, and then inverse-FFT the image. This showed any discrepancies between the two images.



The difference in exposure causes the 'subtraction' of the two images to be an ineffective way of identifying the asteroid. While the asteroid is visible in the difference of the two images, it is dwarfed by the exposure difference on bright stars, even though the two pictures are essentially the same.

However, this method could not account for differences in exposure time, rotation, and scale in both images. Likewise, with our current method, we cannot simply transpose and compare the centroids from both images. Instead, we cross-reference our asteroid candidates with a dim reference star catalogue to address the differences between the two pictures due to exposure inconsistencies.

6.2 Automatically Adapting for Difficult Images and Histograms

In preparing an image for centroids, we first convert the grayscale image into a filtered, binary representation through the thresholding process. However, because images vary in exposure, we cannot filter two different pictures based on same threshold. Pictures with dimmer, fewer stars require a lower threshold, lest their stars get completely filtered out. Pictures with many clearly defined stars require a higher threshold to filter out noise and keep the number of stars reasonable. To account for these "difficult" images and exposure discrepancies during thresholding, our program dynamically iterates through each image and uses a binary search to alter the threshold level in accordance with the number of stars left in the image by the end of each filter iteration. We ultimately arrive at filtered, binary images with appropriate numbers of stars.

Similarly, during the triangle method, we implement multiple histograms in finding the correct scale, rotation, and translation between images and reference catalogs. However, histograms may occur in which the most-occurring value, the peak, is spread out along a wide distribution. The program addresses this issue by adjusting the number of bins for any given parameter based on the distribution of the peak values at the end of each histogram iteration. For more difficult histograms, the program re-histograms the distribution using more bins until it obtains a clear peak value.

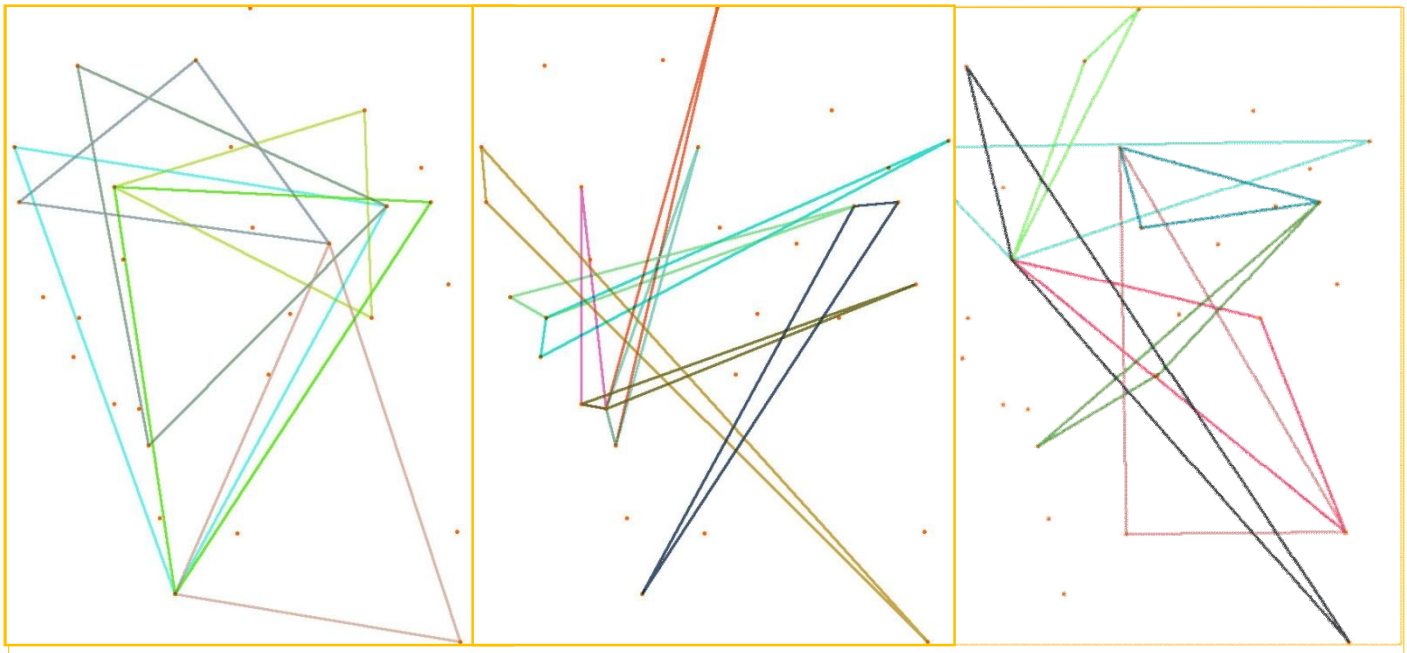
6.3 Improving the Triangle Method

As described in the paper "Asteroid identification at discovery," the triangle method was found to be the most inefficient out of three baseline approaches for locating a picture in the sky. For our purposes, we reasoned that the triangle method would work effectively for our task of localized searches, rather than the whole-sky searches the paper discussed. In light of this, we have implemented the triangle method and incorporated our own insights on the approach.

There are two primary challenges in the triangle method. The first is picking a small number of the same stars in both the image and in the sky for accurate and efficient comparison. Second, identifying 'spurious' triangles that only happen to be similar between the two images. Our attempts to mitigate these problems are detailed below.

6.3.1 Eliminating Certain Triangle Shapes

Not all triangles are easily compared between images. Consider stars forming equilateral triangles in both the image and in the reference. While the computer would easily identify the two triangles as similar, it would be easy for which side is 'longest' to change, causing an incorrect calculation of rotation between those two triangles. Similarly, triangles that are two isosceles, either because they are close to equilateral or because the are excessively scalene, are prone to the same miscalculation of rotation.

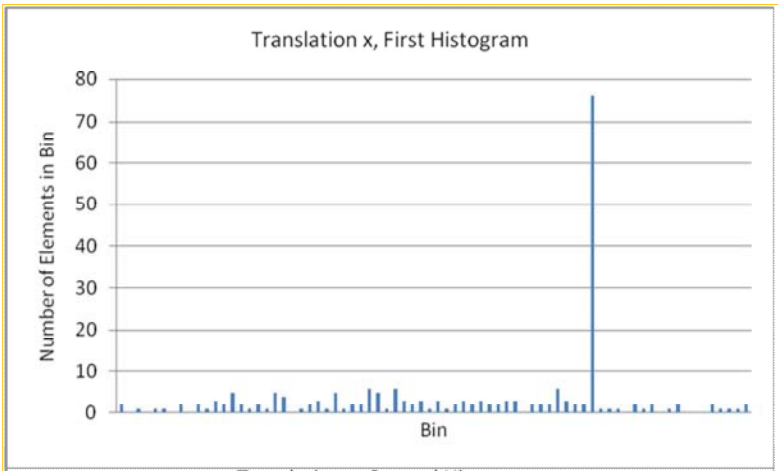


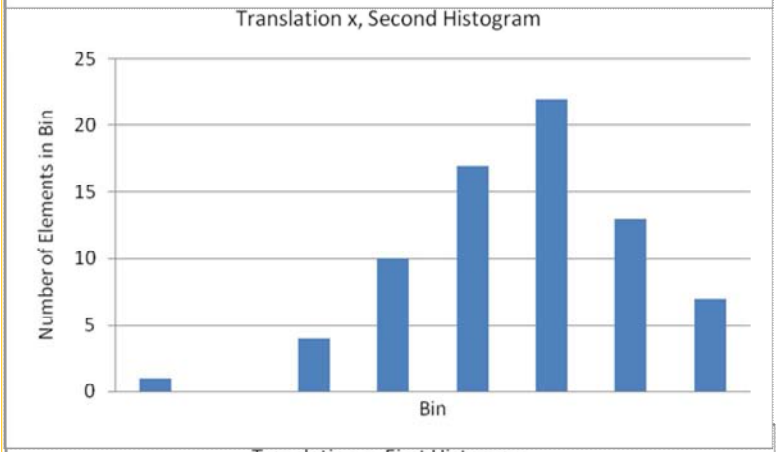
Triangles on the left are too equilateral and are eliminated, as are the ones in the middle, which are too scalene. The triangles to the right have sides that are not easily to confuse, and are used in the analysis.

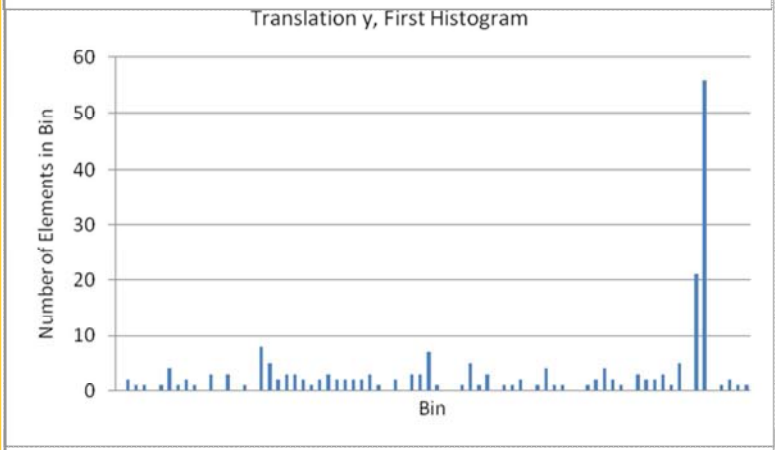
Since many of these triangles would have rotation and translation that is very close to the correct value, they would be especially difficult to eliminate through filtering. Instead, we eliminate these triangles immediately. Above is a picture of the types of triangles we eliminate and types we keep.

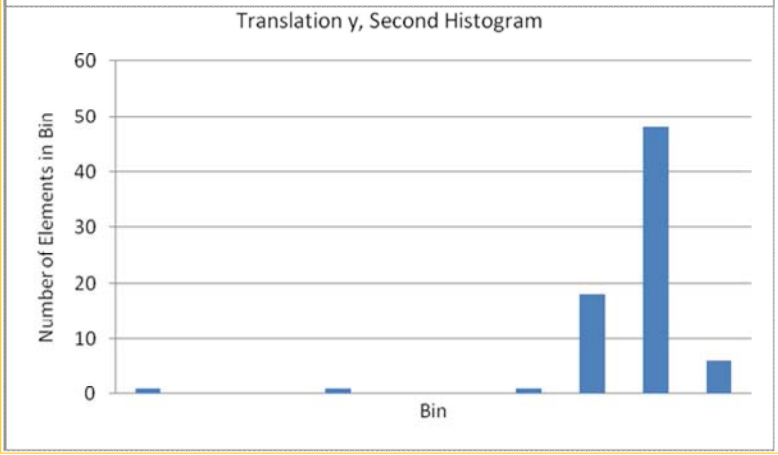
6.3.2 Filtering Triangles

The result of the triangle method to align images is a list of rotations, scales, and translations for the image based on their values for various matching triangles. However, this list is not completely accurate. Sometimes, two triangles will appear to look similar even though they are comprised of completely different stars. The computer will still recognize these triangles as pairs due to their similar length ratios. We want to eliminate the rotation, scale, and translation elements based on these inaccurate triangle pairs. Therefore, it is necessary to “filter” our list of elements to sort out the good triangles from the bad triangles.









These are four graphs of histograms of the x and y translation elements. Each element has a histogram applied twice. For the first element, the peak clearly falls completely within one bin. The elements in this best bin are then re-analyzed in the second histogram. Here the answer seems to fall over multiple bins. To ensure the elimination of all bad triangles, only the two bins on either side of the best bin are then included in the final range of accepted values.
For the y translation histogram, the peak appears to fall between two bins. Both bins are then re-histogramed.

By considering the relative size of multiple bins, the program better accounts for peaks falling between bins, and loses fewer correct triangles in the final answer.

Our original method of filtering was to average all the elements of rotations, scales, and translations. However, this did not work because of the large number of bad triangle matches. Furthermore, this method was extremely susceptible to bad values for the elements. For example, a single scale factor that is an outlier can be very influential on the rest of our data. Therefore, we were forced to devise a better method to filter out the rotations, scales, and translations formed by bad triangle matches. Upon further investigation, we discovered that the correct rotation, scale, and translation between two pictures appear much more frequently in our list than any other value. This is because any correct pair of triangle matches will generate the same, correct rotation, scale, and translation; any incorrect pair will generate an incorrect set of elements that varies from triangle to triangle. This means that if we plot the different values of rotation, scale, and translation, there will be a "peak" in the graph where the correct value lies.

Based on our observations, our current method utilizes a histogram function to identify this peak among the noise of incorrect elements. First, all the values of scale between the two pictures are sorted and distributed among a preset number of bins. Next, the best with the highest frequency is recorded and its frequency is compared with those of the adjacent bins. If the neighboring bins also have a high frequency, it means that the peak may be spread over a bin boundary. Therefore, we also place the neighboring bin into our range for the peak, or the correct scale. To obtain more accuracy in determining the correct scale, we then divide the "best bin" into smaller bins, and repeat the histogram. The result of this process is a narrow range of values in which the correct scale between the two images falls in.

The histogram process is also repeated to find the correct angle and translation. Finally, when all these ranges of values are found, we cross-reference them with the list of all rotations, scales, and translations. In order for a set of elements generated from two triangles to be correct, then it must fall in all of the ranges. Through these criteria, we can drastically cut down the number of triangles pairs to a smaller number of pairs that we can ensure the accuracy of. Throughout this entire process, there are many adjustable variables such as the number of bins we use for the histograms. This is crucial to the method because our analysis images are so different in quality as well as how many stars they have in them. It is important to be able to change these variables to adjust for all the different images that we analyze.

6.3.3 Retaining Multiple Matching Triangles

When trying to find the correct conversion between two sets of points it is important to make sure there are a large number of correct triangles as well as evenly distributed bad triangles. One way to ensure both is to create a list of best matches to one triangle, rather than picking only the best match.

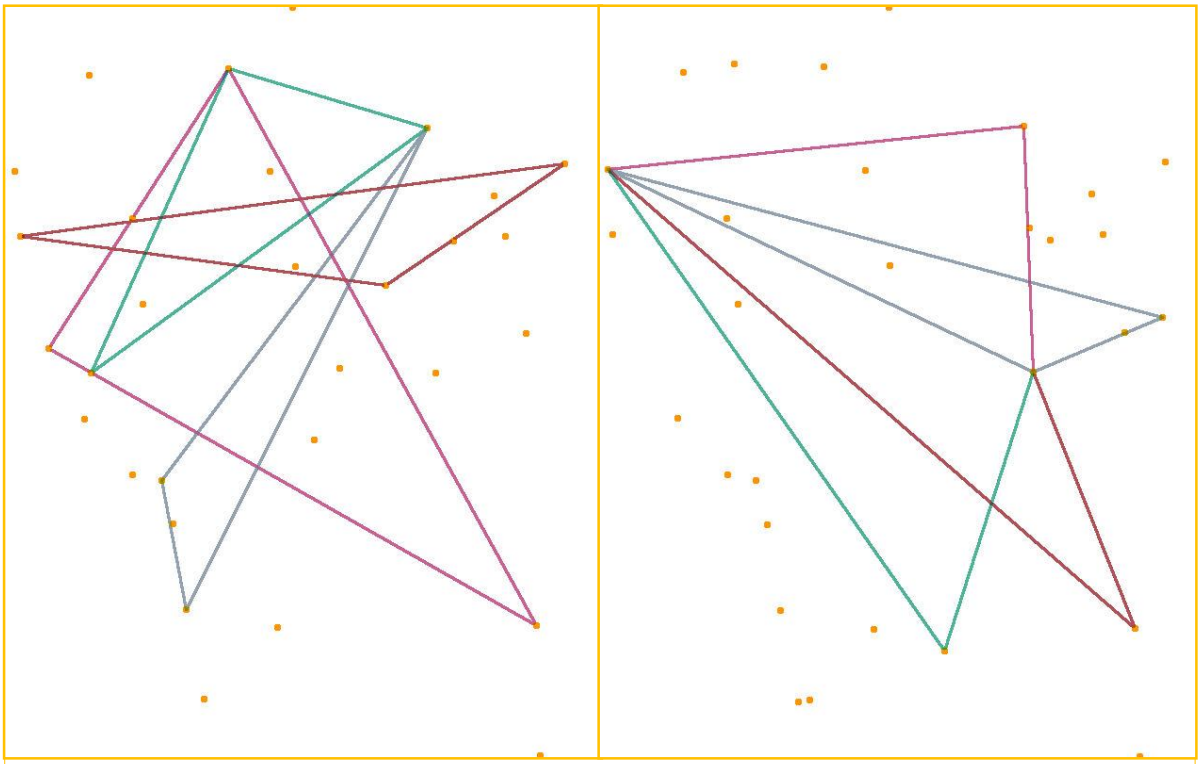
Consider a single triangle in the image. The program will go through all the triangles in the reference and try to find triangles similar to the image triangle. It is possible that the reference stars do not include one of the vertexes of the triangle, in which case the program will not find an exact match. However, it is also possible that a different, incorrect triangle in the

reference will be more similar than the correct corresponding triangle. In this case, if every triangle was matched one-to-one, the correct triangle would be lost. However, when we keep the closest N similar triangles to a correct triangle, we are almost guaranteed that the correct triangle will be included and that all the others will only contribute to evenly distributed noise, increasing the effectiveness of the histogram method.

6.3.4 Signal to Noise Ratio

When we first started to work with the triangle method, we assumed that the overwhelming majority of the similar triangles the program found would be correct. Much to our surprise, this is not necessarily the case. Foremost, it is very difficult to pick the same points to compare in an image and its reference stars, or even between two images' centroids. This decreases the number of possible corresponding triangles and increases the number of non-corresponding triangles that may cause spurious matches. Secondly, in order to immediately eliminate difficult-to-filter bad angles, we ignore certain isosceles triangle shapes. Since we are not sure of the accuracy of the centroids at the start of the analysis, we slightly over-filter out these shapes, reducing the number of potential correct triangles. Finally, keeping more than one similar triangle increases the signal, but also the noise. While all of these partially contribute to a low signal to noise ratio, they also help adapt the method for use with histograms, ultimately improving the final answer.

The more surprising fact is that, given two partially corresponding sets of points, it is easier to form two similar triangles between non-corresponding points than one would expect. In fact, approximately half of all the similar triangles the program identifies are wrong. Below are some examples of such triangles.



The yellow dots are the centroids from two pictures of the same star field. Each of the triangles in the same colors in both images show triangles the program has identified as similar. These triangles need to be filtered out because they are not between corresponding stars.

It is for this reason that the signal to noise ratio is not as high as we initially expected. However, we have still improved on the performance of the triangle method by making these modifications.

6.3.5 Dim Stars

A possible refinement of the results can be made by comparing centroids to a dim star reference. Dim stars tend to be smaller in the picture, so their centroids will be more defined, leading to increased accuracy. Therefore, we devised a method to refine the rotation, scale, and translation generated by comparing centroids to bright stars. We do this by comparing the centroids to the dimmer stars in the reference catalog. Using similar methods as bright stars, we can find a rotation, scale, and translation factor for the images based on their dimmer stars.

As promising as this seems, there is one major problem. Unlike bright stars, we cannot be sure that the dim stars in the reference catalog will be visible in the image. Even if they are visible, they may be so faint that a centroid is not calculated for them. Therefore, there will actually be fewer corresponding stars between the centroids and the reference database. This means fewer correct triangle matches will be generated, which leads to a smaller peak in the distribution of the three elements. Despite more accurate centroids, calculating rotation, scale, and translation based on dim stars actually gives worse results than the elements calculated from

bright stars.

6.3.6 Image Processing Error Analysis

In order to ensure enough accuracy in our image analysis to identify asteroids, we ran tests over several images. Once we obtain scale, rotation, and translation factors between an image and the reference, we then apply these factors to all the pixel coordinates of the stars in the image. This creates a list of RA's and Dec's for all the stars based solely on our image processing. Next, this list of stars coordinates is manually compared to the list of RA's and Dec's given by the reference catalog, and the difference between the two sets of coordinates for a star is calculated in degrees.

The following chart shows the error analysis for one of our test images. The Test RA and Dec are the coordinates generated by our program, while the real RA and Dec are the coordinates from the reference catalog.

Image: 3n1

STAR #	TEST RA	TEST DEC	REAL RA	REAL DEC	DIFF
Star 0	-0.000101	-0.066103	-0.0009738	-0.0653548	0.00115
Star 1	0.038786	-0.065288	0.0374318	-0.0648148	0.001434
Star 2	0.033753	-0.040985	0.032575	-0.040528	0.001264
Star 3	0.011726	-0.073419	0.0107595	-0.072648	0.001236
Star 4	0.056945	-0.079492	0.055573	-0.078531	0.001675
Star 5	-0.017012	0.044111	-0.016866	0.044137	0.000148
Star 6	0.076075	0.056427	0.075424	0.056572	0.000667
Star 7	0.048595	-0.006802	0.047348	-0.00667	0.001254
Star 8	0.039844	0.001476	0.0390103	0.0015342	0.000836
Star 9	-0.009788	0.012226	-0.010391	0.012356	0.000617
Star 10	0.026509	-0.046877	0.025612	-0.04672	0.000911
Star 11	-0.047238	-0.074628	-0.04785	-0.073764	0.001059
Star 12	0.031797	-0.006524	0.0306109	-0.006237	0.00122
Star 13	0.075849	0.01008	0.0745021	0.010037	0.001348
Star 14	0.020711	0.06013	0.019942	0.060056	0.000773

The average difference in coordinates was 0.001039 degrees, which corresponds to 3.440397 pixels on our picture. The most likely source comes from centroid computation. Although we optimize accuracy by weighing the center of the star by brightness, there is still error involved. In particular, bigger, brighter stars have less accuracy in their centroids due to their larger size on the picture. In this image, it is noticeable that Stars 1 through 4 have large errors. This is because the stars are numbered in order of total brightness, and these stars are bigger. Despite this, the 0.001039 degree average error is well within the bounds needed to identify asteroids.

On another image fch2a, we ran another error test, yielding the following results.

Image: fch2a

STAR #	TEST RA	TEST DEC	REAL RA	REAL DEC	DIFF
Star 0	183.125300	3.404217	183.124843	3.404046	0.000488
Star 1	183.150471	3.454929	183.150265	3.454743	0.000278
Star 2	183.157712	3.371486	183.157822	3.371195	0.000311
Star 3	183.322195	3.332647	183.322437	3.332794	0.000284
Star 4	183.294007	3.597640	183.293506	3.597453	0.000535
Star 5	183.139349	3.488501	183.138423	3.488331	0.000941
Star 6	183.146056	3.479500	183.145737	3.478391	0.001154
Star 7	183.142848	3.417571	183.142527	3.417396	0.000365
Star 8	183.164346	3.543978	183.164354	3.543655	0.000323
Star 9	183.107176	3.593695	183.106660	3.592966	0.000894
Star 10	183.332481	3.318382	183.332997	3.318900	0.000730
Star 11	183.158839	3.399388	183.158701	3.399170	0.000258
Star 12	183.274130	3.503729	183.273827	3.503750	0.000303
Star 13	183.176141	3.361482	183.176042	3.361426	0.000114
Star 14	183.299652	3.363647	183.299677	3.363405	0.000244

This picture also yielded extremely accurate results, seeing that the average error for the stars was 0.000481 degrees, which is actually only 0.795041 pixels for this image.

6.4 Orbit Determination on the GPU

In this project, we use the GPU for orbit determination. Given a large number of object observations, we want to find out which of these observations are of the same object. To do this, it is necessary to analyze the orbits such observations would potentially form. First, rough orbits can be calculated and analyzed for whether or not they are reasonable (many orbit determination methods only produce reasonable results for very accurate data). This throws out a huge number of incorrect orbits. The remaining reasonable orbits are then calculated with a more advanced and compute intensive method, such as N-body correction, and the process of verifying these orbits begins.

Normally, for a large number of observations, N, performing the initial rough orbit calculation would involve calculating N choose 3 orbits with the Method of Gauss (which requires three observations), and would take a prohibitively large amount of time. However, this problem has now become a potential candidate for the GPU's large compute power. It requires little data, only six numbers per orbit, describing the time, Ra, and Dec of three observations as well as a single bit that returns whether or not the orbit is reasonable. The Method of Gauss itself involves large amounts of computation, and although it does have two loops, it does not have any other conditional statements. Since the goal is to compute a rough orbit, the fact that the GPU

does not support high-precision variables is not an issue, and in fact the difference high-precision makes in the answer is less than the accuracy that the Method of Gauss provides.

However there are still challenges in calculating orbits on the GPU. Two primary concerns are memory restrictions in the code itself and the specifics of how single-precision effects the calculation, as detailed below.

6.4.1 Optimization for Low-Memory Conditions

While the amount of bandwidth the process requires is very limited because of the small input and output, the amount of memory needed for the calculation itself is large. The orbit determination program itself is essentially a series of algebraic statements, where each variable requires memory from the platform. The size of the calculation makes the memory requirement for calculating an orbit higher than the available register memory for each individual processor on the GPU.

The amount of memory used in the program will also affect runtime. Pipelining is where the GPU loads the data needed for multiple orbits into the memory of a processor. When this data is quickly available the processor spends the least time idle waiting for the next orbit's data, and therefore is most efficient. For this reason, it is beneficial to not crowd each processor's registers with variables pertaining to each calculation, leaving more room for efficient pipelining.

Reducing the memory needed for calculation is done in several ways. Foremost is reusing variables after they are no longer needed. This was done by entering constants directly into the code and using #defines that transform where variables are stored without changing the readability of the code. For example, the following original code

```
//////////////////////////
//EARTH ORBITAL ELEMENTS//
//////////////////////////
double earth_a=1.000732110928368E+00;
double earth_e=1.599910197101524E-02;
double earth_i=1.509328650209302E-03*pi/180;
double earth_omegacap=1.242780078662596E+02*pi/180;
double earth_omegasmall=3.392836473925709E+02*pi/180;
double earth_M2=1.784467221663580E+02*pi/180;
double tmid=2453555.50000;
double teph1=t1;
double teph2=t2;
double teph3=t3;

//////////////////////
//GENERATE R VECTORS//
//////////////////////
double n=k*sqrt(mew/pow(earth_a,3));
double Meph1=n*(teph1-tmid)+earth_M2;
double Meph2=n*(teph2-tmid)+earth_M2;
double Meph3=n*(teph3-tmid)+earth_M2;
```

```
double E1=findE(Meph1, earth_e);
double E2=findE(Meph2, earth_e);
double E3=findE(Meph3, earth_e);

tvec r_orb1=orbitalvector(earth_a,earth_e,E1);
tvec r_orb2=orbitalvector(earth_a,earth_e,E2);
tvec r_orb3=orbitalvector(earth_a,earth_e,E3);

tvec r_ec1=rtoecliptic(r_orb1.x, r_orb1.y, r_orb1.z, earth_omegasmall, earth_i, earth_omegacap);
tvec r_ec2=rtoecliptic(r_orb2.x, r_orb2.y, r_orb2.z, earth_omegasmall, earth_i, earth_omegacap);
tvec r_ec3=rtoecliptic(r_orb3.x, r_orb3.y, r_orb3.z, earth_omegasmall, earth_i, earth_omegacap);

tvec R1=ecliptictoequitorial(r_ec1.x, r_ec1.y, r_ec1.z, epsilon);
tvec R2=ecliptictoequitorial(r_ec2.x, r_ec2.y, r_ec2.z, epsilon);
tvec R3=ecliptictoequitorial(r_ec3.x, r_ec3.y, r_ec3.z, epsilon);
```

Was transformed into the following code

```
#define earth_a              1.000732110928368E+00
#define earth_e              1.599910197101524E-02
#define earth_i              1.509328650209302E-03*pi/180
#define earth_omegacap       1.242780078662596E+02*pi/180
#define earth_omegasmall     3.392836473925709E+02*pi/180
#define earth_M2             1.784467221663580E+02*pi/180
#define tmid                 2453555.50000

#define n w1
        n=k*sqrt(mew/pow3(earth_a));

#define r_orb1 v1
#define r_orb2 v2
#define r_orb3 v3

   r_orb1=orbitalvector(earth_a, earth_e, findE(n*(t1-tmid)+earth_M2, earth_e));
   r_orb2=orbitalvector(earth_a, earth_e, findE(n*(t2-tmid)+earth_M2, earth_e));
   r_orb3=orbitalvector(earth_a, earth_e, findE(n*(t3-tmid)+earth_M2, earth_e));

#undef n
#define r_ec1 v4
#define r_ec2 v5
#define r_ec3 v6

   r_ec1=rtoecliptic(r_orb1.x, r_orb1.y, r_orb1.z, earth_omegasmall, earth_i, earth_omegacap);
   r_ec2=rtoecliptic(r_orb2.x, r_orb2.y, r_orb2.z, earth_omegasmall, earth_i, earth_omegacap);
   r_ec3=rtoecliptic(r_orb3.x, r_orb3.y, r_orb3.z, earth_omegasmall, earth_i, earth_omegacap);

#undef r_orb1
#undef r_orb2
#undef r_orb3
#define R1 v1
#define R2 v2
#define R3 v3
   R1=ecliptictoequitorial(r_ec1.x, r_ec1.y, r_ec1.z, epsilon);
   R2=ecliptictoequitorial(r_ec2.x, r_ec2.y, r_ec2.z, epsilon);
   R3=ecliptictoequitorial(r_ec3.x, r_ec3.y, r_ec3.z, epsilon);
```

```
#undef r_ec1
#undef r_ec2
#undef r_ec3
```

The original code required 17 floats and 9 vectors, a total of 44 floats The new code requires at its peak six vectors, a total of 18 floats, and only retains the answer, three vectors or 9 floats. The remaining three vectors and one float may be reused later in the program. This type of transformation cut the amount of required memory about in half, down to 10 vectors and 26 floats, a total of 56 floats.

Another source of memory usage is from variables in functions the orbit determination uses. While all the functions we wrote specifically for the orbit determination were optimized similarly for minimal memory usage, there are other functions that take up large amounts of memory. Trigometric functions and power functions were two huge sources of memory usage. In fact, at one point, all the trig functions were #defined into square roots just to get the program to run at all! To fix this, all the powers were moved out of the power function and into expanded form. So pow(x,6) became x*x*x*x*x*x. This alone reduced the runtime of the program by half, as well as reducing the amount of memory needed.

The reason for the effect of power functions is that they use complex algorithms to accommodate non-integer powers. Examining the assembly language reveals some of effect. Multiplying numbers together continuously calls the “mul” operator multiple times, while doing the same thing with the power function stars declaring variables and moving memory around, performing very few actual calculations.

These optimizations were still not enough to comfortably run the program on the GPU. By further moving some variables out of the processors’ registers and into shared memory finally allowed the large number of variables used in the orbit determination program to be run successfully on the GPU. Further optimizations are in progress, both to reduce the amount of needed memory as well as more selectively picking which variables should be in shared memory for efficient runtime.

6.4.2 Optimization for Low-Precision Conditions

Single-precision calculations do not have a significant effect on the answer of any given orbit, but they can effect runtime. In the orbit determination, it is necessary to find roots of an eighth order polynomial, in this case using Newton’s method. This involves calculating the value of a function at a ‘guess’ and using a calculated derivative to move your guess closer to the root until it is within some desired accuracy. The original code for this process is below:

```
float equation(float a,float b,float c,float r2){
   return pow(r2,8)+a*pow(r2,6)+b*pow(r2,3)+c;
}

float derv (float a,float b,float c,float r2){
```

```
    return 8*pow(r2,7)+6*a*pow(r2,5)+3*b*pow(r2,2);
}

float newton (float a,float b,float c,float userguess){
    float slope=derv (a,b,c,userguess);
    float yintercept=equation(a,b,c,userguess)-(slope)*userguess;
    float xintercept=-1*yintercept/slope;
    return xintercept;
}

    float guess = 3;
    float answer=newton(a, b, c, guess);
while (abs((newton(a, b, c, answer)-answer))>=0.000001)
          answer=newton(a,b,c,answer);
    float r2=answer;
```

We eventually added a counter to the loop and found that it had trouble converging when using single precision but not double precision. The most important reason for this is the equation and derv functions. When using single precision, these functions will add up rounded versions of each of these numbers raised to a power. By modifying the code as shown below, the calculation's accuracy increases merely by changing the order of operations so as not to round as often.

```
float equation( float a, float b, float c, float r2){
    return ((pow2(r2)+a)*pow3(r2)+b)*pow3(r2)+c;
}

float derv ( float a, float b, float c, float r2){
    return ((8*pow2(r2)+6*a)*pow3(r2)+3*b)*pow2(r2);
}
```

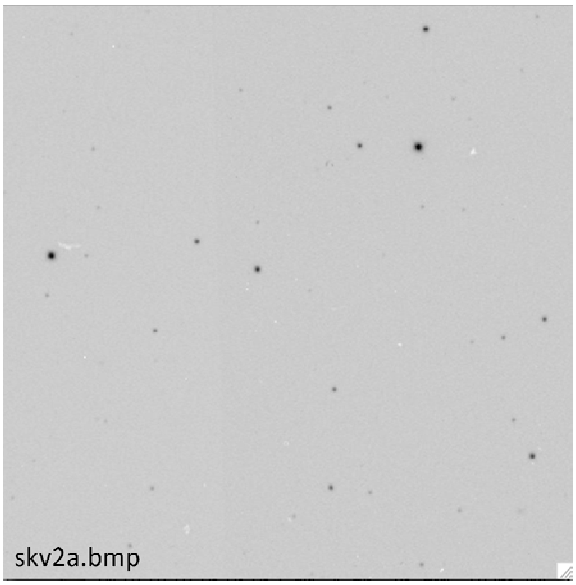
This change saves memory, loops fewer times, and decreases runtime by increasing the accuracy of the calculation.

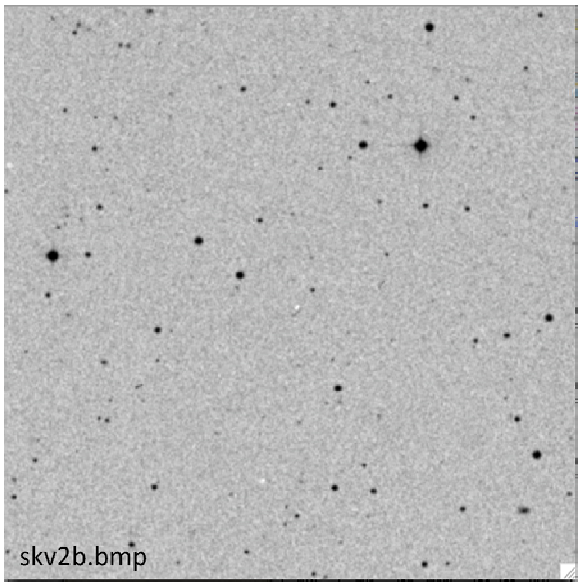
6.4.3 Current GPU Analysis

These optimizations have allowed us to run the orbital determination on the GPU despite stringent memory constraints. Since the number of orbits needed to be calculated is N choose 3, for large N the overhead for loading memory into the GPU will be negligible. Even considering the overhead, initial performance timers are encouraging, performing as well as the CPU on small test data sets. We will optimize the GPUs performance in the future, specifically focusing on where memory is located.

7. Sample Results

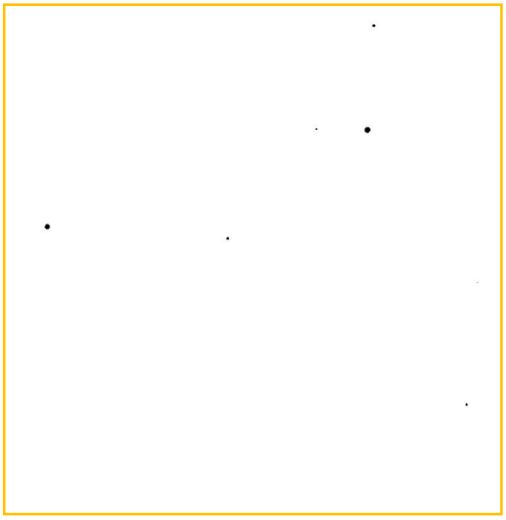
We begin with two images, skv2a.bmp and skv2b.bmp.





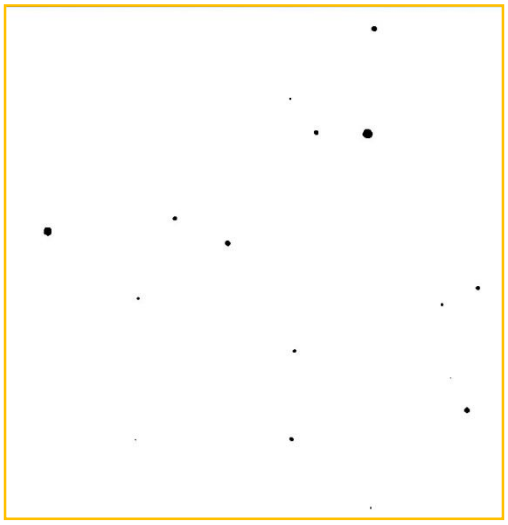
Although these images are of the same approximate region in the sky, each was taken at a different time. Furthermore, skv2a is from the Near-earth Asteroid Tracking Service while the other image, skv2b, is a reference spectra image. Therefore there exists a major exposure difference between the two. This creates an issue for image processing because we cannot filter both images the same way. To account for these exposure discrepancies, our image analysis code iterates through both images until it finds a unique, reasonable binary threshold for each image based on the number of stars.

For skv2a, the image itself is much dimmer, so the program takes 3iterations in computing the acceptable threshold.

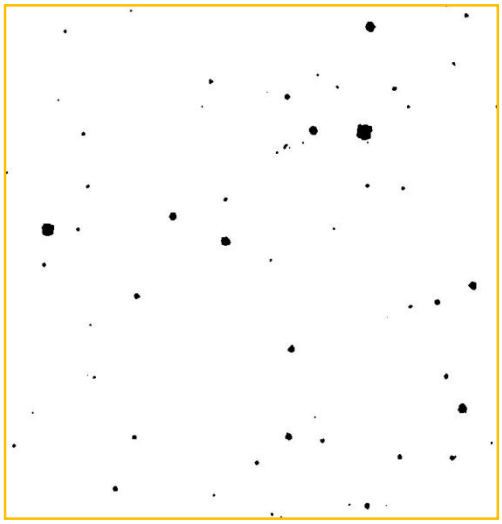


After the first filter iteration with the default threshold setting, the image retains only 6

stars. Since this star number is less than the minimum required number of stars, the threshold is lowered through a binary search that converges on the new, more correct threshold.

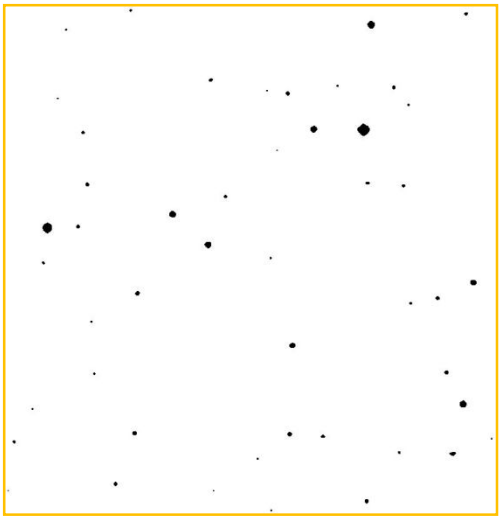


After the second pass with the new threshold value, we arrive at an image that retains 15 stars, a number that still does not meet the minimum required number of stars. Thus, the program again conducts a second binary search for a more appropriate threshold.



After the third filter iteration using the new, lower threshold value, we finally arrive at an image with 58 stars that easily passes the minimum star number requirement. Thus, the thresholding iterations stop for skv2a.

For skv2b, the image is already much more defined than skv2a, so the program more easily finds a viable threshold.



The default threshold value is enough to filter the image, leaving 48 stars.

After thresholding, centroids are calculated. This returns a list of stars and the corresponding X-Y pixel center for each.

```
star 1:
        (387.816772, 131.072495)star 2:
        (44.393642, 231.909149)star 3:
        (237.388046, 244.449341)star 4:
        (494.797729, 418.486969)star 5:
        (394.993561, 21.548277)star 6:
        (333.096802, 129.886230)star 7:
        (180.982101, 218.391449)star 8:
        (306.391876, 447.569489)star 9:
        (309.476654, 356.513885)star 10:
        (506.563843, 291.570984)star 11:
        (468.000000, 308.000000)current starnum is 13
```

The program then performs the triangle method, finding a conversion factor between the images and between the X-Y coordinates and the RA-DEC celestial coordinates. In doing so, the program filters triangles through multiple histogram bins for scale, rotation, and translation. For example, we can see there is a clear peak bin for this parameter.

```
 Largest peak: 1117, second largest 162. Printing bins:
0 0 0 0 1 0 1 0 0 0 0 1 0 1 0 1 0 0 0 0 1 0 0 1 0 0 1 0 0 0 1 0 2 1 1 0 2 1 1 1 0 1 3 0 2 3 3 1 2
 3 1 7 3 7 5 5 10 15 11 13 20 20 28 23 29 49 42 39 69 73 84 124 137 1117 162 129 110 124 117 99 1
25 135 136 126 125 105 121 97 108 124 104 102 106 88 89 87 52 63 48 43 39 34 29 26 16 8 11 10 11
13 8 4 4 4 5 2 3 1 4 2 3 2 1 0 0 0 1 0 5 2 0 0 2 1 0 0 0 0 1 1 0 1 1 0 0 0 2 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
```

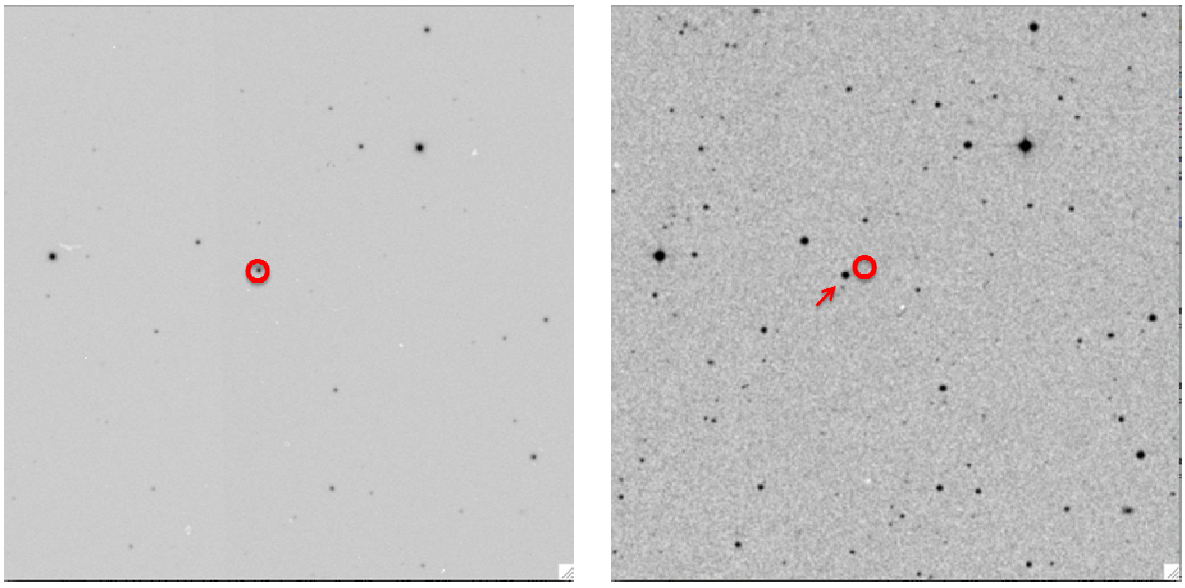
Then, the program uses the conversion factor obtained from the triangle method to transpose the centroids from one image to another. This obtains possible asteroid candidates, which we then cross-reference with the dim star catalog to verify these candidates are indeed asteroids and not dim stars or the result of exposure differences.

```
ImageA->ImageB::Scale=0.998993,Angle=0.002020,trans=<0.218310,1.156407>
ImageB->DimReference::Scale=0.000695,Angle=3.571987,trans=<229.611173,22.883553>
```

Lastly, the program prints out our asteroid and its location in both X-Y and RA-DEC coordinates.

```
Possible Asteroid at XY(328.893707,377.689485) RADEC(229.760678,23.049279)
```

Referring back to our original two observational images, we can see that the program has successfully identified an asteroid that has moved from one image to another.



Note that this method works for images from completely different times- these two images only happened to have objects in similar locations. This makes our analysis very different from traditional forms of asteroid identification, which primarily look for the movement of objects.

8. Future Work

This project is still in progress. In the future we would like to:

- Apply our program to a larger quantity of data.
- Further improve our methods of image analysis. Particularly, we would like to develop better methods of choosing the same stars in the image and in the reference catalog. This would greatly improve the results of the triangle method for difficult star fields.
- Apply our GPU orbital determination program to a larger number of asteroid observations. Constructing sets that contain known asteroids can give us an understanding of how useful this type of analysis will be when applied to real data.
- Improve and analyze the performance of our programs. There are several optimizations that can be made to the image analysis program, especially for larger numbers of stars. Reducing the memory needed for the GPU program would also aid its performance.

9. Conclusion

We have created a conceptual framework to explore and analyze longitudinal astronomical image data. We have developed systems to find asteroids in images and locate these images in the sky. In the process, we implemented enhancements to the Triangle Method to increase its accuracy. These enhancements are unique to our work.

We have begun to use the GPU (graphical processing unit) to analyze which observations of unidentified objects could be of the same asteroid. This process involves running orbit determination on a GPU, and is also original to our work.

10. Acknowledgements

We would like to thank our mentors, Dr. Erik DeBenedictis, Dr. Danhong Huang, and Dr. Qin Hong, for their guidance in all areas of the project.

Two researchers in this project attended Summer Science Program at New Mexico Tech in the summer of 2008. Our mentors from this program, Dr. Kevin Krisciunas, Dr. William Anderson, and Dr. Agnes Kim, taught us valuable skills and have continued to provide guidance for this project throughout the year.

Dr. Lorie Liebrock, of New Mexico Tech, provided valuable input for our efforts to host this project on the GPU.

We would especially like to acknowledge the Albuquerque Astronomical Society (TAAS) for their guidance and facilities. We were able to gain valuable hands-on experience with observational astronomy at the General Nathan Twining Observatory, a facility of TAAS in Belen, New Mexico. A TAAS member, Bill Wallace, has also provided us with observational images for analysis.

Our thanks also go to Dr. Eileen Ryan of Magdalena Ridge Observatory for her advice on locating more astronomical images for anlaysis. She put us in contact with Robert Holmes, of the Astronomical Research Institute, who provides astronomical data for educational purposes.

We would finally like to thank Beverly DeBenedictis for editorial and presentation assistance. Her help has been invaluable. Thank you!

11. Bibliography

11.1 Online Data

"ESO Archive Query Form." ESO/ST-ECF — SAF - Science Archive Facility . 28 Feb. 2009 <http://archive.eso.org/eso/eso_archive_main.html>.

“HORIZONS System." JPL Solar System Dynamics. 28 Feb. 2009 <http://ssd.jpl.nasa.gov/?horizons>.

"VizieR Service." VizieR Service. 28 Feb. 2009 <http://vizier.u-strasbg.fr/viz-bin/VizieR>.

"USNO Image and Catalogue Archive." USNO Flagstaff Station . 28 Feb. 2009 <http://www.nofs.navy.mil/data/FchPix/>.

"SkyView Virtual Observatory." National Aeronautics and Space Administration . 19 Mar. 2009 <http://skyview.gsfc.nasa.gov/cgi-bin/titlepage.pl>.

"Near Earth Objects" Astronomical Research Institute . 19 Mar. 2009 <http://ari.home.mchsi.com>.

11.2 Software

"CCfits: C++ Wrappers for the cfitsio library." HEASARC: NASA's Archive of Data on Energetic Phenomena. 28 Feb. 2009 <http://heasarc.gsfc.nasa.gov/fitsio/CCfits/>.

Baer, Jim. "Comet/asteroid Orbit Determination and Ephemeris Software." CODES. 28 Feb. 2009 <home.earthlink.net/~jimbaer1/

Tschumperlé, David. "The CImg Library - C++ Template Image Processing Toolkit." The CImg Library - C++ Template Image Processing Toolkit. 8 Mar. 2009 <http://cimg.sourceforge.net/>

11.3 In the News

Brown, Dwanye. "NASA Statement on Student Asteroid Calculations." 16 Apr. 2008. NASA. 31 Mar. 2009<http://www.nasa.gov/home/hqnews/2008/apr/HQ_08103_student_asteroid_calculations.html>.

Schmitt, Stephen R. "Computing planet positions using mean orbital elements." 2005. 31 Mar. 2009 <http://home.att.net/~srschmitt/planetorbits.html>.

Technology Review, Robert. "Technology Review: Giant Camera Tracks Asteroids ." Technology Review: The Authority on the Future of Technology. 28 Feb. 2009 <http://www.technologyreview.com/computing/21705/?a=f>.

"The Albuquerque Astronomical Society TAAS." Redirect. 28 Feb. 2009 <http://www.taas.org/taas/index.php>

"Current Impact Risks." Near-Earth Object Program. 28 Feb. 2009 <http://neo.jpl.nasa.gov/risk/>.

11.4 Papers and Books

“Asteroid identification at discovery,” Mikael Granvik, 2003, Master's Theisis, University of Helsinki. Padgett, Curtis, Kenneth Kreutz-Delgado, and Suraphol Udomkesmalee.

"Evaluation of Star Identification Techniques." Journal of Guidance, Control, and Dynamics 20, No. 2.March-April (1007): 259-267.

A. Milani, M. E. Sansaturio, and S. R. Chesley, “The Asteroid Identification Problem IV: Attributions,” *Icarus* 151, no. 2 (2001): 150-159.

Scientific Detectors for Astronomy 2005 : Explorers of the Photon Odyssey (Astrophysics and Space Science Library) (Astrophysics and Space Science Library). New York: Springer, 2006.

The scientist and engineer's guide to digital signal processing, Smith SW, California Technical Publishing San Diego, CA, USA, 1997

NVIDIA CUDA Compute Unified Device Architecture Programming Guide Version 1.0, NVIDIA Corporation, 6/23/2007

Green, Bill . "Histogram, Thresholding and Image Centroid Tutorial." Drexel University - Unix Web Service . 8 Mar. 2009 <http://www.pages.drexel.edu/~weg22/hist_thresh_cent.html>

"Image Analysis - Connected Components Labeling." Informatics Homepages Server. 8 Mar. 2009 <http://homepages.inf.ed.ac.uk/rbf/HIPR2/label.htm>

Shutler , Jamie. "Cartesian moments." Informatics Homepages Server. 15 Aug. 2002. 8 Mar. 2009 <http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/SHUTLER3/node4.html

12. Appendix

In the following sections, we provide a guide to the code, its functions, and how it incorporates the methods we have described in our report. All of the code we have developed is included after its overview.

12.1 Online Interfacing Guide - C++

The online interface code takes a FITS file image, looks up the reference stars for that image, and writes all the data into a form we can use in the next program. First, the RA and DEC are obtained from the header of the FITS file. The program constructs a hyperlink to the query the VizieR online star database with the Ra and Dec. The program then opens a connection to the website, uses the hyperlink that has been created, and downloads the HTML page the website returns with the `DBaccess` function.

Each star catalog we query returns different information, so parsing the webpage has separate cases for each. First, hyperlinks are stripped from the webpage in the `deleathyperlinks` function. We assign a pointer to the position of the beginning of each catalog, and then use this information to decide how to parse stars at different locations in the file. This puts all the star information in the `RefStars` array.

Some locations will have stars that are right next to each other but have coordinates on the 0-360 break. We correct the stars' positions for this, favoring negative angles over large separations.

Finally, we print a file that begins with the number of stars in the image and lines containing each stars' Ra, Dec, proper motion, magnitude, and what catalog they were from. This file is used in the next processing stage.

12.2 Online Interfacing Code

```
#include "CImg.h"
#include <stdio.h>
#include <stdlib.h>
#include "stdafx.h" // must be present for Visual C++. For Unix, stdafx.h can be empty
#include <math.h>
#include <string.h>
#include <iostream>
#include <fstream>
#include <string>

using namespace cimg_library;
using namespace std;

#define THRESHOLD_ADJUST 65
#define MAX_STAR_NUM 500
#define MAX_STAR_SIZE 900
#define NUMREFSTARS 15
#define NUMIMGSTARS 20
```

```
#define NUMKEEP 8
#define INTRODUCEDERROR 0.0
#define TESTDATAMODE 0 //set to 1 to manually rotate and scale a set of reference stars

//Erika stuff
#define _CRT_SECURE_NO_WARNINGS 1
#define _CRT_NONSTDC_NO_WARNINGS 1

extern int mStartup();
extern char* DBaccess(char*);
extern int mCleanup();

#define pi 3.14159265358979323846 2643
#define MAXSTARSIMG 1000

struct Star{
double RA;
double DEC;
int alivebit; //deleated or no?
double vMAG;
double pmRA;
double pmDEC;
int checknum;
int catalogtype; //0 for bright, 1 for all
};

struct vector{
double a;
double b;
double c;
};

struct pointvec{
vector a;
vector b;
vector c;
};

double spaceatof(char *pointer){
while (*pointer==' ')
pointer++;
return atof(pointer);
}

void deleathyperlinks(char *page){
char *rpointer=page;
char *wpointer=page;
while(*rpointer!=0){
if (*rpointer!='<')
*wpointer++ = *rpointer++;
else if (strncmp(rpointer, "</EM></A>", 9) == 0)
for (int i = 0; i < 9; i++) *wpointer++ = *rpointer++;
else if(rpointer[0]=='<' && rpointer[1]=='/' && rpointer[2]=='A' && rpointer[3]=='>')
rpointer+=4;
else if(strncmp(rpointer, "<A HREF=", 8) == 0) {
while (*rpointer!=0 && *rpointer !='>')
```

```
rpointer++;
if (*rpointer=='>') rpointer++;
}
else
*wpointer++ = *rpointer++;
}
*wpointer = 0;
}

//pixel coordinates and brightness
typedef struct {
int x;
int y;
int val;
} PIX;

int cmp(const void* av, const void *bv){ //sorting thing
double *a = (double *)av;
double *p1 = (double *)bv;
if (*a>*p1)
return 1;
if (*a==*p1)
return 0;
else
return -1;
}

int main() {
/***
Reference Star Section- Preprocessing
***/
double RAdeg=0;
double DECdeg=0;
int arcminwid=0;
int arcminhgt=0;
ofstream fout ("starsa1.out"); //file to write the stars to when done parsing
ifstream fin ("starsa1.in"); //file to read for approximate RA and DEC from FITS
fin >> RAdeg;
fin >> DECdeg;
fin >> arcminwid;
fin >> arcminhgt;

printf("hello world\n");

//internet access version
char iAddress[300];
sprintf(iAddress, "/viz-bin/VizieR?-source=I/289,I/294,I/280A,I/297&-c=%.6f%+.6f&-c.bm=%dx%d&-
out.max=100&-out.form=;SV", RAdeg, DECdeg, arcminwid, arcminhgt);
printf("url is %s\n", iAddress);
mStartup();
char* bar=DBaccess(iAddress);

char *UCACC, *UCACB, *ASC, *NOMAD;
deleathyperlinks(bar);
//pointer to beginnning of UCAC2 Catalogue (Zacharias+ 2003) The Second U.S. Naval Observatory CCD
Astrograph Catalog (48330571 rows)
```

```
UCACC =strstr(bar, "UCAC2 C");
//pointer to beginnning of UCAC2 Bright Star Supplement (Urban+, 2004) UCAC Bright Star Supplement (430000 rows)
UCACB =strstr(bar, "UCAC2 B");
//pointer to beginnning of All-sky Compiled Catalogue of 2.5 million stars (Kharchenko 2001) The all-sky catalogue of 2.5million stars (2501313 rows)
ASC =strstr(bar, "All-sky C");
//pointer to beginnning of All-sky Compiled Catalogue of 2.5 million stars (Kharchenko 2001) The all-sky catalogue of 2.5million stars (2501313 rows)
NOMAD=strstr(bar, "NOMAD");

Star RefStars[MAX_STAR_NUM];
int starnum=0;
char *here; //pointer to current position
here=bar;
while ((here=strstr(here, "</EM></A>"))!=0){
char *fieldstart, *raplace, *decplace, *magplace, *pmraplace, *pmdecplace;
int typebit=-1;
if(UCACB > here && here > UCACC){ //UCAC2 Catalogue
fieldstart = here+9;
raplace = fieldstart + 12;
decplace = fieldstart + 25;
magplace = fieldstart + 46;
pmraplace = fieldstart + 62;
pmdecplace = fieldstart + 74;
typebit=0;
}
else if(ASC > here && here > UCACB){ //UCAC2 Bright Star Supplement
fieldstart = here+9;
raplace = fieldstart + 12;
decplace = fieldstart + 26;
magplace = fieldstart + 59;
pmraplace = fieldstart + 70;
pmdecplace = fieldstart + 78;
typebit=0;
}
else if(here > ASC){ //All-sky Compiled Catalogue
fieldstart = here+9;
raplace = fieldstart + 2;
decplace = fieldstart + 16;
pmraplace = fieldstart + 48;
pmdecplace = fieldstart + 58;
magplace = fieldstart + 74;
typebit=0;
}
else{
printf("VERY BADDDDDDDD!!!! Help with looking up reference stars parsing thing!\n");
}
if(here>NOMAD){ //NOMAD
fieldstart=here+29;
raplace=fieldstart;
decplace=fieldstart+13;
pmraplace=fieldstart+32;
pmdecplace=fieldstart+49;
magplace=fieldstart+62;
typebit=1;
```

```
}
RefStars[starnum].RA=atof(raplace); //adding on a degree DELEATEEEEEEEEEEEEEEEE!!!!!!!!!!!!!!!!
RefStars[starnum].DEC=atof(decplace);
RefStars[starnum].vMAG=spaceatof(magplace);
RefStars[starnum].pmRA=spaceatof(pmraplace);
RefStars[starnum].pmDEC=spaceatof(pmdecplace);
RefStars[starnum].checknum=starnum;
RefStars[starnum].alivebit=1;
RefStars[starnum].catalogtype=typebit;
starnum++;
here+=9;
}

/*if(starnum>NUMREFSTARS){
//pick the top NUMREFSTARS brightest stars
double sortit0[MAX_STAR_NUM];
for(int i=0; i<starnum; i++){
sortit0[i]=RefStars[i].vMAG;
}
qsort((void*) sortit0, starnum, sizeof(double), &cmp);
double maxmag=sortit0[NUMREFSTARS];
for(int i=0; i<starnum; i++){
if(RefStars[i].vMAG>maxmag)
RefStars[i].alivebit=0;
}
}*/
//Making sure we don't have a 0-360 overlay problem.
double sortit1[MAX_STAR_NUM];
double sortit2[MAX_STAR_NUM];
for(int i=0; i<starnum; i++){
sortit1[i]=RefStars[i].RA;
sortit2[i]=RefStars[i].DEC;
}
qsort((void*) sortit1, starnum, sizeof(double), &cmp);
qsort((void*) sortit2, starnum, sizeof(double), &cmp);
if (sortit1[starnum-2]-sortit1[0] > 180){
for(int i=0; i<starnum; i++){
if (RefStars[i].RA > 180){
RefStars[i].RA=RefStars[i].RA - 360;
}
}
}
if (sortit2[starnum-2]-sortit2[0] > 180){
for(int i=0; i<starnum; i++){
if (RefStars[i].DEC > 180){
RefStars[i].DEC=RefStars[i].DEC - 360;
}
}
}

//mCleanup();
printf("closed internet \n");
delete bar;
/*
Format:
number of stars #
```

```
each line has Ra Dec pmRA pmDEC vMAg catalog -all with spaces in between
*/
fout << starnum << endl;
for(int i=0; i< starnum; i++){
fout << RefStars[i].RA << " " << RefStars[i].DEC << " " << RefStars[i].pmRA << " " << RefStars[i].pmDEC << " "
<< RefStars[i].vMAG << " " << RefStars[i].catalogtype << endl;
}
printf("ending program\n");
return 0;

}
```

12.3 Image Analysis Guide - C++

The image analysis portion of the code extracts possible asteroid data using a set of two corresponding images and reference star catalogs. Image analysis itself is separated into several major parts. Centroid calculation is the first major part. The `centroid` function requires a bitmap image input and it outputs the number of stars in the image and a STAR array. The STAR structure stores the area and brightness of each star, a PIX array which includes the x-y coordinates and brightness of each pixel in each star, and a CENTROID structure, which contains its own set of centroid x-y coordinates. In preparing the image for centroids, each image is filtered and made binary through a thresholding process. The program automatically compensates for exposure differences on different images by dynamically adjusting the threshold value based on the number of stars in any given image.

We perform `centroid` on two images, A and B. The function returns two corresponding STAR arrays, ImageStarsA and ImageStarsB. Using the resulting ImageStars arrays, which contain the centroids for each image, we then implement the `FindTriangle` method to align the images and find a conversion factor from x-y coordinate system to the RA-DEC coordinate system. In filtering for viable triangles, `getHistogram` narrows down our matching triangles based on a distribution of rotation, scale, and translation “bins”. Like it does for image thresholding, the program dynamically adjusts the number of bins for each parameter based on the distribution of the most occurring “peak” values, ultimately arriving at a clearly defined peak value. `FindTriangle` is first used image-to-image so we can obtain a conversion factor from ImageStarsA to ImageStarsB. Then, we loop though both images and transpose each centroid from ImageStarsB onto ImageStarsA, obtaining a new ImageStarsC. After comparing the distances between the centroids of ImgStarsC, any centroid in the image that does not lie within a reasonable distance from another centroid is considered to be a potential asteroid. Using the results from the triangle method, we can determine the RA-DEC celestial coordinates for each potential asteroid. The `FindTriangle` is then used two more times: ImageStarsB to the (bright) reference stars, RefStars, and ImageStarsB to the dim stars, dimRefStars. This essentially compares the results from our image-to-image analysis with actual stars in the sky.

The last portion of image analysis determines whether or not a centroid is an asteroid. A possible asteroid from the image-to-image analysis is cross-referenced with stars in dimRefStars.

If a centroid in dimRefStars corresponds to the asteroid candidate, then the program prints the centroid as a dim star. If not, then the centroid is declared an asteroid and its x-y and RA-DEC coordinates are printed.

12.4 Image Analysis Code

```
#include "CImg.h"
#include <stdio.h>
#include <stdlib.h>
//#include "stdafx.h" // must be present for Visual C++. For Unix, stdafx.h can be empty
#include <math.h>
#include <string.h>
#include <iostream>
#include <fstream>
#include <string>

using namespace cimg_library;
using namespace std;

#define WANTEDSTARNUM 110 //min number of stars a picture must have
#define NUMROOM 10 //how far away it can be from WANTEDSTARNUM
#define DIIFICULTIMAGE 0 //if 1 goes through thresholds more slowly, without binary search
#define MAXREPEAT 9 //max number of times you go through centroids
#define INITIALTHRES 80 //initial try for threshold
#define MAXTHRES 190 //maximum threshold
#define MAX_STAR_NUM 2001
#define MAX_STAR_SIZE 900
#define NUMREFSTARS 40
#define NUMIMGSTARS 40
#define NUMKEEP 4
#define INTRODUCEDERROR 0.0
#define TESTDATAMODE 0 //set to 1 to manually rotate and scale a set of reference stars
#define FIRSTBINS 200 //200 original. Sets tolerance, lower for higher tolerance. DO NOT change!
#define MAXLOOP 10 //how many times to loop the histograms max, no more than 19
#define CEN_PERFORM 0 //weather or not to add extra timers in the centroid function

//Erika stuff
#define _CRT_SECURE_NO_WARNINGS 1
#define _CRT_NONSTDC_NO_WARNINGS 1

extern int mStartup();
extern char* DBaccess(char*);
extern int mCleanup();

#define pi 3.14159265358979323846264 3
#define MAXSTARSIMG 4000 //1000
#define MAXBINNUM 100

void magicalbreakpoint(const char *t) {
printf("\n");
printf(t);
printf("\n");
}
```

```
#ifdef _DEBUG
#define ASSERT(c) if (!(c)) magicalbreakpoint("Assert failed " #c "\n");
#else
#define ASSERT(c) ;
#endif
struct Star{
double RA;
double DEC;
int alivebit; //deleated or no?
double vMAG;
double pmRA;
double pmDEC;
int checknum;
int catalogtype; //0 for bright, 1 for all
};

class StarArray {
unsigned int Size;
Star *Base;
//ImgStarsB[MAX_STAR_NUM];
public:
StarArray(unsigned int s) {
Size = s;
Base = new Star[s];
}
~StarArray() {
delete Base;
}
Star &operator[](unsigned int index) {
ASSERT(index < Size);
return Base[index];
}
};

struct vector{
double a;
double b;
double c;
};

struct pointvec{
vector a;
vector b;
vector c;
};

struct triangle{
vector pos1;
vector pos2;
vector pos3;
pointvec numberedstars; //ordered pair of pos num of star opposite smallest to largest side
int check1;
int check2;
int check3;
double d12; //distance between stars
double d23;
```

```
double d31;
double sd; //small, medium, large relative side lengths
double md;
double ld;
int CorrTri[NUMKEEP]; //corresponding triangle in other list
double currdiff[NUMKEEP]; //current difference between corresponding triangle and yourself
//factors for img->ref
double scale[NUMKEEP];
double angle[NUMKEEP];
vector displacement[NUMKEEP];
int onoff[NUMKEEP]; //1 if triangle is beign included for consideration in end rotation, scale, translation
};

class trianglearray{
unsigned int size;
triangle *data;
triangle **adata;
//triangle foobar[MAXSTARSIMG*5];
public:
trianglearray(){
size=10;
data=new triangle[10];
}
~trianglearray(){
delete data;
}
triangle &operator[](unsigned int index) {
if(index>=size) {
triangle *newdata=new triangle[index*3/2];
ASSERT(newdata);
for(unsigned int i=0; i<size; i++)
newdata[i]=data[i];
printf("growing to %d\n", index*3/2);
delete data;
data=newdata;
size=index*3/2;
}
return data[index];
}
};

double spaceatof(char *pointer){
while (*pointer==' ')
pointer++;
return atof(pointer);
}

void deleathyperlinks(char *page){
char *rpointer=page;
char *wpointer=page;
while(*rpointer!=0){
if (*rpointer!='<')
*wpointer++ = *rpointer++;
else if (strncmp(rpointer, "</EM></A>", 9) == 0)
for (int i = 0; i < 9; i++) *wpointer++ = *rpointer++;
else if(rpointer[0]=='<' && rpointer[1]=='/' && rpointer[2]=='A' && rpointer[3]=='>')
```

```
rpointer+=4;
else if(strncmp(rpointer, "<A HREF=", 8) == 0) {
while (*rpointer!=0 && *rpointer !='>')
rpointer++;
if (*rpointer=='>') rpointer++;
}
else
*wpointer++ = *rpointer++;
}
*wpointer = 0;
}

vector sort3(vector me){
double Small=me.a;
double Medium=me.b;
double Large=me.c;
while(Small>Medium || Medium>Large || Small>Large){
if(Small>Medium){
double hold=Medium;
Medium=Small;
Small=hold;
}
if(Medium>Large){
double hold=Large;
Large=Medium;
Medium=hold;
}
}
vector sorted;
sorted.a=Small;
sorted.b=Medium;
sorted.c=Large;
return sorted;
}

void SetTri(triangle *Tri, Star &a, Star &b, Star &c, double error = 0.){
Tri->pos1.a=a.RA + error*rand()/32767; Tri->pos1.b=a.DEC + error*rand()/32767;
Tri->pos2.a=b.RA + error*rand()/32767; Tri->pos2.b=b.DEC + error*rand()/32767;
Tri->pos3.a=c.RA + error*rand()/32767; Tri->pos3.b=c.DEC + error*rand()/32767;
Tri->d12= sqrt((Tri->pos1.a-Tri->pos2.a)*(Tri->pos1.a-Tri->pos2.a)+(Tri->pos1.b-Tri->pos2.b)*(Tri->pos1.b-Tri->pos2.b));
Tri->d23= sqrt((Tri->pos2.a-Tri->pos3.a)*(Tri->pos2.a-Tri->pos3.a)+(Tri->pos2.b-Tri->pos3.b)*(Tri->pos2.b-Tri->pos3.b));
Tri->d31= sqrt((Tri->pos3.a-Tri->pos1.a)*(Tri->pos3.a-Tri->pos1.a)+(Tri->pos3.b-Tri->pos1.b)*(Tri->pos3.b-Tri->pos1.b));
vector sort1;
sort1.a=Tri->d12;
sort1.b=Tri->d23;
sort1.c=Tri->d31;
sort1=sort3(sort1);
Tri->sd=1;
Tri->md=sort1.b/sort1.a;
Tri->ld=sort1.c/sort1.a;

if (Tri->d12==sort1.a)
Tri->numberedstars.a=Tri->pos3;
```

```
else if (Tri->d23==sort1.a)
Tri->numberedstars.a=Tri->pos1;
else if (Tri->d31==sort1.a)
Tri->numberedstars.a=Tri->pos2;

if (Tri->d12==sort1.b)
Tri->numberedstars.b=Tri->pos3;
else if (Tri->d23==sort1.b)
Tri->numberedstars.b=Tri->pos1;
else if (Tri->d31==sort1.b)
Tri->numberedstars.b=Tri->pos2;

if (Tri->d12==sort1.c)
Tri->numberedstars.c=Tri->pos3;
else if (Tri->d23==sort1.c)
Tri->numberedstars.c=Tri->pos1;
else if (Tri->d31==sort1.c)
Tri->numberedstars.c=Tri->pos2;

for(int i=0; i<NUMKEEP; i++){
Tri->CorrTri[i]=-2;
Tri->currdiff[i]=100;
Tri->onoff[i]=1;
}

vector sort2;
sort2.a=a.checknum;
sort2.b=b.checknum;
sort2.c=c.checknum;
sort2=sort3(sort2);
Tri->check1=(int)sort2.a;
Tri->check2=(int)sort2.b;
Tri->check3=(int)sort2.c;
}

void FindTranslation(triangle *Img, triangle *Ref, int i){
//scale
vector sortImg;
sortImg.a=Img->d12;
sortImg.b=Img->d23;
sortImg.c=Img->d31;
sortImg=sort3(sortImg);
vector sortRef;
sortRef.a=Ref->d12;
sortRef.b=Ref->d23;
sortRef.c=Ref->d31;
sortRef=sort3(sortRef);
double scaleImgtoRef=sortRef.c/sortImg.c;
Img->scale[i]=scaleImgtoRef;

//rotation
vector ImgLSide;
ImgLSide.a=Img->numberedstars.a.a - Img->numberedstars.b.a;
ImgLSide.b=Img->numberedstars.a.b - Img->numberedstars.b.b;
double ImgAngle=atan2(ImgLSide.b, ImgLSide.a);
if(ImgAngle<0)
```

```
ImgAngle+=2*pi;
vector RefLSide;
RefLSide.a=Ref->numberedstars.a.a - Ref->numberedstars.b.a;
RefLSide.b=Ref->numberedstars.a.b - Ref->numberedstars.b.b;
double RefAngle=atan2(RefLSide.b, RefLSide.a);
if(RefAngle<0)
RefAngle+=2*pi;
double Theta=RefAngle-ImgAngle;
if (Theta<0)
Theta+=2*pi;
Img->angle[i]=Theta;

//translation
double newRA= (cos(Theta)*Img->numberedstars.a.a-sin(Theta)*Img->numberedstars.a.b)*scaleImgtoRef;
double newDec=(sin(Theta)*Img->numberedstars.a.a+cos(Theta)*Img->numberedstars.a.b)*scaleImgtoRef;
vector translate;
translate.a=Ref->numberedstars.a.a-newRA;
translate.b=Ref->numberedstars.a.b-newDec;
Img->displacement[i]=translate;
if(translate.a<-100000){
int waaaaa=1;
}
}

void XYtoRADEC(double x, double y, double scale, double rotation, double translationra, double translationdec,
double *answera, double *answerb){
double newRA= (cos(rotation)*x-sin(rotation)*y)*scale;
double newDEC=(sin(rotation)*x+cos(rotation)*y)*scale;
newRA+=translationra;
newDEC+=translationdec;
*answera=newRA;
*answerb=newDEC;
}

//Back to Chris code
//star centroid
typedef struct {
double x;
double y;
} CENTROID;

//pixel coordinates and brightness
typedef struct {
int x;
int y;
int val;
} PIX;

//star
typedef struct {
PIX pixels[MAX_STAR_SIZE];
int area;
double brightness;
CENTROID cent;
} STAR;
```

```
int cmp(const void* av, const void *bv){ //sorting thing
double *a = (double *)av;
double *p1 = (double *)bv;
if (*a>*p1)
return 1;
if (*a==*p1)
return 0;
else
return -1;
}

void getHistogram(int *bins, int binNum, double *sortit, int newtrinum, double *sortit2, int *bestbinfill, int
ifangle=0)
{
//decimal percent. if a bin adjacent to the best bin exceeds this percent of the best bin it is also included
double perct=.2;
//reset array of bins
for (int i = 0; i < 2000; i++) {
bins[i] = 0;
}

//deciding the width of each bin
double binwidth = ((sortit[newtrinum - 1] - sortit[0]) / binNum);
double binleft = sortit[0];
double binright = binleft + binwidth;

//look at elements to fill bins. Now we have the histogram
for (int bin = 0; bin < binNum; bin++) {
for (int i = 0; i < newtrinum; i++) {
if (sortit[i] <= binright && sortit[i] > binleft) {
bins[bin]++;
}
}
binleft = binleft + (binwidth);
binright = binright + (binwidth);
}

int bestbin = -1;
int counter2 = 0;
int NumInBestBin=0;

//find the best bin. NumInBestBin consistantly will be the original number of elements in the single best bin
for (int i = 0; i < binNum; i++) {
if (bins[i] > *bestbinfill) {
bestbin = i;
*bestbinfill=NumInBestBin = bins[i];
}
}
//check to see if the bin above should also be included in peak
if(bestbin<newtrinum)
if(bins[bestbin+1] > (int)(perct*NumInBestBin))
*bestbinfill+=bins[bestbin+1];
//whole mod 2pi exception, patching up between first and last bin- only for angle
if(ifangle){
//if the best bin is last, do I include the first bin as well?
if(bestbin==binNum-1 && bins[0] > (int)(perct*NumInBestBin)){
```

```
*bestbinfill+=bins[0];
for(int i=0; i<newtrinum; i++){ //subtract 2pi from everythign over pi
if(sortit[i]>pi){
sortit[i]-=2*pi;
}
}
}

//if the best bin is first, so I include the last bin as well?
if(bestbin==0 && bins[binNum-1] > (int)(perct*NumInBestBin)){
*bestbinfill+=bins[binNum-1];
for(int i=0; i<newtrinum; i++){ //subtract 2pi from everythign over pi
if(sortit[i]>pi){
sortit[i]-=2*pi;
}
}
}
}

//standard non-wrapping lower bound
if(bestbin>0 && bins[bestbin-1] > (int)(perct*NumInBestBin) ){
*bestbinfill+=bins[bestbin-1];
for (int i = 0; i < bestbin-1; i++) //counter2=number of elements before the (best bin-1)
counter2 += bins[i];
}
else{
for (int i = 0; i < bestbin; i++) //counter2=number of elements before the best bin
counter2 += bins[i];
}

//add elements in the best bin to the outgoing array
for (int i = 0; i < *bestbinfill; i++) {
sortit2[i] = sortit[counter2 + i];
}

//add on last bin elements for angle wrapping
if(ifangle && bestbin==0){
int NumBefWrap=*bestbinfill-bins[binNum-1];
for (int i = 0; i < binNum-1; i++)
counter2 += bins[i];
for (int i = counter2; i < newtrinum-1; i++) {
sortit2[NumBefWrap+i-counter2] = sortit[i];
}
}
//add on first bin elements for angle wrapping
if(ifangle && bestbin==binNum-1){
int NumBefWrap=*bestbinfill-bins[0];
for (int i = 0; i < bins[0]; i++) {
sortit2[NumBefWrap+i] = sortit[i];
}
}
}

//prints out histogram and checks if there are multiple sizeable peaks
void checkhistogram(int *counter, int size, int printbit=0){
int largepeak=0;
```

```
int largepeak2=0; //second largest peak
for(int i=0; i<size; i++){
if(counter[i]>largepeak){
largepeak2=largepeak;
largepeak=counter[i];
}
else if(counter[i]>largepeak2)
largepeak2=counter[i];
}
if(printbit){
printf("\n Largest peak: %d, second largest %d. Printing bins:\n", largepeak, largepeak2);
for(int i=0; i<size; i++){
printf("%d ", counter[i]);
}
printf("\n");
}
}

//takes the triangle data from BTri[i].[j], accesses PointsA,B, and plots triangles.
void printtriangles(StarArray &PointsB, int numpointb, StarArray &PointsA, int numpointa, trianglearray &ATri,
trianglearray &BTri, int i, int j,
double RAsubtract=0, double DECsubtract=0, double mult=1.5){
int Wid = 422*3/2, Hei = 532*3/2;
CImg<int> imgd0a(Wid, Hei, 1, 3, 255);
CImg<int> imgd0b(Wid, Hei, 1, 3, 255);
const unsigned char color1[] = { 255, 150, 1, };
for(int a=0; a<numpointa; a++){
imgd0a.draw_circle((int)((PointsA[a].RA-RAsubtract)*mult), (int)((PointsA[a].DEC-DECsubtract)*mult), 2,
color1, 1);
}
for(int a=0; a<numpointb; a++){
imgd0b.draw_circle((int)(PointsB[a].RA*3/2), (int)(PointsB[a].DEC*3/2), 2, color1, 1);
}

CImgList<int> points;
points.insert(CImg<int>::vector((int)((ATri[BTri[i].CorrTri[j]].pos1.a-RAsubtract)*mult),
(int)((ATri[BTri[i].CorrTri[j]].pos1.b-DECsubtract)*mult)));
points.insert(CImg<int>::vector((int)((ATri[BTri[i].CorrTri[j]].pos2.a-RAsubtract)*mult),
(int)((ATri[BTri[i].CorrTri[j]].pos2.b-DECsubtract)*mult)));
points.insert(CImg<int>::vector((int)((ATri[BTri[i].CorrTri[j]].pos3.a-RAsubtract)*mult),
(int)((ATri[BTri[i].CorrTri[j]].pos3.b-DECsubtract)*mult)));
points.insert(CImg<int>::vector((int)((ATri[BTri[i].CorrTri[j]].pos1.a-RAsubtract)*mult),
(int)((ATri[BTri[i].CorrTri[j]].pos1.b-DECsubtract)*mult)));
unsigned char color[] = { 1, 255, 1, }; color[0] = rand(); color[1] = rand(); color[2] = rand();
imgd0a.draw_line(points, color, (float).8);
CImgList<int> points2;
points2.insert(CImg<int>::vector((int)(BTri[i].pos1.a*3/2), (int)(BTri[i].pos1.b*3/2)));
points2.insert(CImg<int>::vector((int)(BTri[i].pos2.a*3/2), (int)(BTri[i].pos2.b*3/2)));
points2.insert(CImg<int>::vector((int)(BTri[i].pos3.a*3/2), (int)(BTri[i].pos3.b*3/2)));
points2.insert(CImg<int>::vector((int)(BTri[i].pos1.a*3/2), (int)(BTri[i].pos1.b*3/2)));
imgd0b.draw_line(points2, color, (float).8);
CImgDisplay d1(imgd0a,"a corresponds", 2), d2(imgd0b,"b corresponds", 2);
while (!d1.is_closed && !d2.is_closed) {
CImgDisplay::wait(d1, d2);
}
}
```

```
//gives translation from parameter 1 to parameter 2, that is, PointsB->PointsA
void FindTriangle(StarArray &PointsB, int numpointb, StarArray &PointsA, int numpointa, int catalogtype,
double *nScale, double *nAngle, double *nTransx, double *nTransy,
double Scaletomatch=0, double Angletomatch=0, double Transxtomatch=0, double Transytomatch=0,
double RAsubtract=0, double DECsubtract=0, double mult=1.5){
trianglearray ATri;
trianglearray BTri;
int trinum1=0;
int trinum2=0;

int Wid = 422*3/2, Hei = 532*3/2;
int num = 8;
CImg<int> imgd0a(Wid, Hei, 1, 3, 255);
CImg<int> imgd1a(Wid, Hei, 1, 3, 255);
CImg<int> imgd2a(Wid, Hei, 1, 3, 255);
CImg<int> imgd3a(Wid, Hei, 1, 3, 255);
CImg<int> imgd0b(Wid, Hei, 1, 3, 255);
CImg<int> imgd1b(Wid, Hei, 1, 3, 255);
CImg<int> imgd2b(Wid, Hei, 1, 3, 255);
CImg<int> imgd3b(Wid, Hei, 1, 3, 255);

int trianglecount1=0;
int trianglecount2=0;
int trianglecount3=0;

for(int a=0; a<numpointa-1; a++){
if(PointsA[a].alivebit==1)
for(int b=a+1; b<numpointa-1; b++){
if(PointsA[b].alivebit==1)
for(int c=b+1; c<numpointa-1; c++){
if(PointsA[c].alivebit==1)
if(1){
SetTri(&ATri[trinum1], PointsA[a], PointsA[b], PointsA[c]);
CImgList<int> points;
points.insert(CImg<int>::vector((int)((ATri[trinum1].pos1.a-RAsubtract)*mult), (int)((ATri[trinum1].pos1.b-
DECsubtract)*mult)));
points.insert(CImg<int>::vector((int)((ATri[trinum1].pos2.a-RAsubtract)*mult), (int)((ATri[trinum1].pos2.b-
DECsubtract)*mult)));
points.insert(CImg<int>::vector((int)((ATri[trinum1].pos3.a-RAsubtract)*mult), (int)((ATri[trinum1].pos3.b-
DECsubtract)*mult)));
points.insert(CImg<int>::vector((int)((ATri[trinum1].pos1.a-RAsubtract)*mult), (int)((ATri[trinum1].pos1.b-
DECsubtract)*mult)));
unsigned char color[] = { 1, 255, 1, }; color[0] = rand(); color[1] = rand(); color[2] = rand();
if (ATri[trinum1].ld < 1.5) {
if(trianglecount1==num){
imgd1a.draw_line(points, color, (float).8);
trianglecount1=0;
}
trianglecount1++;
}
else if (ATri[trinum1].ld>8){
if(trianglecount2==num){
imgd2a.draw_line(points, color, (float).8);
trianglecount2=0;
}
```

```
trianglecount2++;
}
else {
if(trianglecount3==num){
imgd3a.draw_line(points, color, (float).8); //changeback!
trianglecount3=0;
}
trianglecount3++;
//imgd.draw_line(points, color, .3);
trinum1++;
}
}
}
}
}

trianglecount1=0; trianglecount2=0; trianglecount3=0;
for(int a=0; a<numpointb-1; a++){
if(PointsB[a].alivebit==1)
for(int b=a+1; b<numpointb-1; b++){
if(PointsB[b].alivebit==1)
for(int c=b+1; c<numpointb-1; c++){
if(PointsB[c].alivebit==1)
if(1){
SetTri(&BTri[trinum2], PointsB[a], PointsB[b], PointsB[c], INTRODUCEDERROR);
CImgList<int> points;
points.insert(CImg<int>::vector((int)(BTri[trinum2].pos1.a*3/2), (int)(BTri[trinum2].pos1.b*3/2)));
points.insert(CImg<int>::vector((int)(BTri[trinum2].pos2.a*3/2), (int)(BTri[trinum2].pos2.b*3/2)));
points.insert(CImg<int>::vector((int)(BTri[trinum2].pos3.a*3/2), (int)(BTri[trinum2].pos3.b*3/2)));
points.insert(CImg<int>::vector((int)(BTri[trinum2].pos1.a*3/2), (int)(BTri[trinum2].pos1.b*3/2)));
unsigned char color[] = { 1, 255, 1, }; color[0] = rand(); color[1] = rand(); color[2] = rand();
if (BTri[trinum2].ld < 1.5) {
if(trianglecount1<num)
imgd1b.draw_line(points, color, (float).8);
trianglecount1++;
}
else if (BTri[trinum2].ld>8){
if(trianglecount2<num)
imgd2b.draw_line(points, color, (float).8);
trianglecount2++;
}
else {
if(trianglecount3<num)
imgd3b.draw_line(points, color, (float).8);
trianglecount3++;
//imgd.draw_line(points, color, .3);
trinum2++;
}
}
}
}
}

//printing out examples of triangles ignored because they were too scalene, equalateral, OK triangles, and just stars
if(0){
const unsigned char color[] = { 255, 100, 1, };
```

```
for(int i=0; i<numpointa; i++){
imgd0a.draw_circle((int)((PointsA[i].RA-RAsubtract)*mult), (int)((PointsA[i].DEC-DECsubtract)*mult), 2, color,
1);
imgd1a.draw_circle((int)((PointsA[i].RA-RAsubtract)*mult), (int)((PointsA[i].DEC-DECsubtract)*mult), 1, color,
1);
imgd2a.draw_circle((int)((PointsA[i].RA-RAsubtract)*mult), (int)((PointsA[i].DEC-DECsubtract)*mult), 1, color,
1);
imgd3a.draw_circle((int)((PointsA[i].RA-RAsubtract)*mult), (int)((PointsA[i].DEC-DECsubtract)*mult), 1, color,
1);
}
for(int i=0; i<numpointb; i++){
imgd0b.draw_circle((int)(PointsB[i].RA*3/2), (int)(PointsB[i].DEC*3/2), 2, color, 1);
imgd1b.draw_circle((int)(PointsB[i].RA*3/2), (int)(PointsB[i].DEC*3/2), 1, color, 1);
imgd2b.draw_circle((int)(PointsB[i].RA*3/2), (int)(PointsB[i].DEC*3/2), 1, color, 1);
imgd3b.draw_circle((int)(PointsB[i].RA*3/2), (int)(PointsB[i].DEC*3/2), 1, color, 1);
}
CImgDisplay d1(imgd1a,"a <2", 2), d2(imgd2a,"a >10", 2), d3(imgd3a,"a OK", 2), d4(imgd1b,"b <2", 2),
d5(imgd2b,"b >10", 2), d6(imgd3b,"b OK", 2), d7(imgd0a,"a dots", 2), d8(imgd0b,"b dots", 2);
while (!d1.is_closed && !d2.is_closed && !d3.is_closed && !d4.is_closed && !d5.is_closed && !d6.is_closed &&
!d7.is_closed && !d8.is_closed) {
CImgDisplay::wait(d1, d4);
CImgDisplay::wait(d2, d5);
CImgDisplay::wait(d3, d6);
CImgDisplay::wait(d7, d8);
}
}

for(int b=0; b<trinum2; b++){
for (int a=0; a<trinum1; a++){
double diff=sqrt((ATri[a].md-BTri[b].md)*(ATri[a].md-BTri[b].md)+(ATri[a].ld-BTri[b].ld)*(ATri[a].ld-
BTri[b].ld));
int tempa=a;
for(int i=0; i<NUMKEEP; i++){
if (diff<BTri[b].currdiff[i]){
int movea=BTri[b].CorrTri[i];
double movediff=BTri[b].currdiff[i];
BTri[b].CorrTri[i]=tempa;
BTri[b].currdiff[i]=diff;
tempa=movea;
diff=movediff;
}
}
}
for(int i=0; i<NUMKEEP; i++){
FindTranslation(&BTri[b], &ATri[BTri[b].CorrTri[i]], i);
}
}

double asortit[MAXSTARSIMG*20];
double ssortit[MAXSTARSIMG*20];
double trasortit[MAXSTARSIMG*20];
double tdecsortit[MAXSTARSIMG*20];

int newtrinum=0;
for(int i=0; i<trinum2; i++){
for(int j=0; j<NUMKEEP; j++)
```

```
if(BTri[i].currdiff[j]<.01 && newtrinum < MAXSTARSIMG*20){
asortit[newtrinum]=BTri[i].angle[j];
ssortit[newtrinum]=BTri[i].scale[j];
trasortit[newtrinum]=BTri[i].displacement[j].a;
tdecsortit[newtrinum]=BTri[i].displacement[j].b;
BTri[i].onoff[j]=1;
if(0) if(BTri[i].currdiff[j]<.01){
printtriangles(PointsB, numpointb, PointsA, numpointa, ATri, BTri, i, j, RAsubtract, DECsubtract, mult);
}
newtrinum++;
}
else
BTri[i].onoff[j]=0;
}

qsort((void*) asortit, newtrinum, sizeof(double), &cmp);
qsort((void*) ssortit, newtrinum, sizeof(double), &cmp);
qsort((void*) trasortit, newtrinum, sizeof(double), &cmp);
qsort((void*) tdecsortit, newtrinum, sizeof(double), &cmp);

int acounter[50];
for(int i=0; i<50; i++){
acounter[i]=0;
}
int scounter[50];
for(int i=0; i<50; i++){
scounter[i]=0;
}
int counter[2000];

double numbertosum=0;
int numberRepeats=0;
while (numbertosum==0 && numberRepeats<MAXLOOP){
double ahigher = 0, alower = 0, shigher = 0, slower = 0, trahigher = 0, tralower = 0, tdechigher = 0, tdeclower = 0;
//Filtering for Bright Stars
if (catalogtype==0){
int numbins1=FIRSTBINS-10*numberRepeats;
int numbins2=10;
if(numbins1<20)
numbins2=18-numberRepeats;

if(0) for(int i=0; i<newtrinum; i++){
printf("anglesort=%f, scalesort=%f\n",asortit[i], ssortit[i]);
}
double asortit2[MAXSTARSIMG*20], ssortit2[MAXSTARSIMG*20], trasortit2[MAXSTARSIMG*20],
tdecsortit2[MAXSTARSIMG*20];
int abestbinfill = -1, sbestbinfill = -1, trabestbinfill = -1, tdecbestbinfill = -1, abestbinfill2 = -1, sbestbinfill2 = -1,
trabestbinfill2 = -1, tdecbestbinfill2 = -1;
int anglebit=0;
if (numberRepeats==1) anglebit=1;
getHistogram(counter, numbins1, asortit, newtrinum, asortit2, &abestbinfill, anglebit);
checkhistogram(counter, numbins1, 1);
if(anglebit)
qsort((void*) asortit, newtrinum, sizeof(double), &cmp);
qsort((void*) asortit2, abestbinfill, sizeof(double), &cmp);
getHistogram(counter, numbins1, ssortit, newtrinum, ssortit2, &sbestbinfill);
```

```
checkhistogram(counter, numbins1, 1);
getHistogram(counter, numbins1, trasortit, newtrinum, trasortit2, &trabestbinfill);
checkhistogram(counter, numbins1, 1);
getHistogram(counter, numbins1, tdecsortit, newtrinum, tdecsortit2, &tdecbestbinfill);
checkhistogram(counter, numbins1, 1);

double asortitbest[MAXSTARSIMG*20], ssortitbest[MAXSTARSIMG*20], trasortitbest[MAXSTARSIMG*20],
tdecsortitbest[MAXSTARSIMG*20];
getHistogram(counter, numbins2, asortit2, abestbinfill, asortitbest, &abestbinfill2);
checkhistogram(counter, numbins2, 1);
getHistogram(counter, numbins2, ssortit2, sbestbinfill, ssortitbest, &sbestbinfill2);
checkhistogram(counter, numbins2, 1);
getHistogram(counter, numbins2*2, trasortit2, trabestbinfill, trasortitbest, &trabestbinfill2);
checkhistogram(counter, numbins2*2, 1);
getHistogram(counter, numbins2*2, tdecsortit2, tdecbestbinfill, tdecsortitbest, &tdecbestbinfill2);
checkhistogram(counter, numbins2*2, 1);

alower = asortitbest[0];
ahigher = asortitbest[abestbinfill2-1];
slower = ssortitbest[0];
shigher = ssortitbest[sbestbinfill2-1];
tralower = trasortitbest[0];
trahigher = trasortitbest[trabestbinfill2-1];
tdeclower = tdecsortitbest[0];
tdechigher = tdecsortitbest[tdecbestbinfill2-1];

for(int i=0; i<trinum2; i++){
for(int j=0; j<NUMKEEP; j++){
if(1) if (BTri[i].angle[j]<alower || BTri[i].angle[j]>ahigher){
BTri[i].onoff[j]=0;
}
if(1) if (BTri[i].scale[j]<slower || BTri[i].scale[j]>shigher){
BTri[i].onoff[j]=0;
}
if(1) if (BTri[i].displacement[j].a<tralower || BTri[i].displacement[j].a>trahigher){
BTri[i].onoff[j]=0;
}
if(1) if (BTri[i].displacement[j].b<tdeclower || BTri[i].displacement[j].b>tdechigher){
BTri[i].onoff[j]=0;
}
}
}
}
//Filtering for Dim Stars
else {
ahigher = Angletomatch*1.05;
alower = Angletomatch*0.95;
shigher = Scaletomatch*1.05;
slower = Scaletomatch*0.95;
trahigher = Transxtomatch*1.05;
tralower = Transxtomatch*0.95;
tdechigher = Transytomatch*1.05;
tdeclower = Transytomatch*0.95;

for(int i=0; i<trinum2; i++){
for(int j=0; j<NUMKEEP; j++){
```

```
if(1) if (BTri[i].angle[j]<alower || BTri[i].angle[j]>ahigher){
BTri[i].onoff[j]=0;
}
if(1) if (BTri[i].scale[j]<slower || BTri[i].scale[j]>shigher){
BTri[i].onoff[j]=0;
}
if(0) if (BTri[i].displacement[j].a<tralower || BTri[i].displacement[j].a>trahigher){
BTri[i].onoff[j]=0;
}
if(0) if (BTri[i].displacement[j].b<tdeclower || BTri[i].displacement[j].b>tdechigher){
BTri[i].onoff[j]=0;
}
}
}
}


double sumscale=0, sumangle=0, sumtransra=0, sumtransdec=0;
for(int b=0; b<trinum2; b++){
for(int j=0; j<NUMKEEP; j++){
//printf("d=%f,s=%f,a=%f,t=<%f,%f>\n", BTri[b].currdiff[j], BTri[b].scale[j], BTri[b].angle[j],
BTri[b].displacement[j].a, BTri[b].displacement[j].b);
if(BTri[b].onoff[j]==1){
sumscale+=BTri[b].scale[j];
sumangle+=BTri[b].angle[j];
sumtransra+=BTri[b].displacement[j].a;
sumtransdec+=BTri[b].displacement[j].b;
//printf("d=%f,s=%f,a=%f,t=<%f,%f>\n", BTri[b].currdiff[j], BTri[b].scale[j], BTri[b].angle[j],
BTri[b].displacement[j].a, BTri[b].displacement[j].b);
//**********************
//HERE IS THE FANCY PRINTING TRIANGLE FUNCTION TOOOOOONY ************************
printtriangles(PointsB, numpointb, PointsA, numpointa, ATri, BTri, b, j, RAsubtract, DECsubtract, mult);
//**********************
numbertosum++;
/* For when doing test data
if(((BTri[b].check1!=ATri[BTri[b].CorrTri[j]].check1)
||(BTri[b].check2!=ATri[BTri[b].CorrTri[j]].check2)||(BTri[b].check3!=ATri[BTri[b].CorrTri[j]].check3))){
printf("-------Incorrect Counted match:\n");
printf("d=%f,s=%f,a=%f\n", ImgTri[b].currdiff[j], ImgTri[b].scale[j], ImgTri[b].angle[j]);
printf("%d:%d:%d\n", ImgTri[b].check1, ImgTri[b].check2, ImgTri[b].check3);
printf("%d:%d:%d\n", RefTri[ImgTri[b].CorrTri[j]].check1, RefTri[ImgTri[b].CorrTri[j]].check2,
RefTri[ImgTri[b].CorrTri[j]].check3);
printf("-------\n");
printf("* ");
}
else{
printf(" ");
}
*/
}
//else
//printtriangles(PointsB, numpointb, PointsA, numpointa, ATri, BTri, b, j, RAsubtract, DECsubtract, mult);
}
}
*nScale=sumscale/numbertosum;
*nAngle=sumangle/numbertosum;
```

```
*nTransx=sumtransra/numbertosum;
*nTransy=sumtransdec/numbertosum;
if(numbertosum==0){
printf("difficult to histogram\n");
*nScale=(shigher+slower)/2;
*nAngle=(ahigher+alower)/2;
*nTransx=(trahigher+tralower)/2;
*nTransy=(tdechigher+tdeclower)/2;
}
printf("\n");
numberRepeats++;
}
}

int centroid(char *imgName, int *numStar, STAR starCent[]) {
LARGE_INTEGER start, end, freq, start2, end2, freq2, start3, end3, freq3, start4, end4, freq4;
CImg<unsigned char> img(imgName);
img=img.get_channel(0);
CImg<unsigned char> bin(532,535);
CImg<unsigned char> bin2(532,535);
STAR stars[MAX_STAR_NUM];
int starNum=0; //total number of stars
int thres = 0;
int cnt = 0; //total number of pixels in a star
PIX pixel;
float xf, yf; //float x and y coordinates for centroid computation
float brightness; //total brightness
int tmp;
STAR tmpStar;


int numrepeat=0;
int thresmax=MAXTHRES;
int thresmin=1;

do { //adjust for images with less brightness
starNum=0;
thres=(int)(thresmax+thresmin)/2;
if(numrepeat==0)
thres=INITIALTHRES;
if(DIIFICULTIMAGE)
thres=INITIALTHRES-30*numrepeat;
if(1) for(int i=0; i< MAX_STAR_NUM; i++){
stars[i].area=0;
stars[i].brightness=0;
}
if(CEN_PERFORM)
QueryPerformanceCounter(&start);

//Convert the image into binary
//img.display();
bin = img.get_threshold(thres);

//bin.display("bin");
if(CEN_PERFORM){
QueryPerformanceCounter(&end);
```

```
QueryPerformanceFrequency(&freq);
QueryPerformanceCounter(&start2);
}

/*Filter,Remove noise. 'Shrink' function.
bin2 = bin;
if (1) for (int i=1; i<bin2.dimx()-1; i++) {
for (int j=1; j<bin2.dimy()-1; j++) {
int a = bin(i, j);
int b = bin2(i,j)=bin(i+1,j)|bin(i-1,j)|bin(i,j+1)|bin(i,j-1)|bin(i,j);
double fd = 213.;
}
}*/

bin2 = bin.get_blur_median();

//bin2.display("filtered");

bin2 = bin2.get_blur_median();
bin2 = bin2.get_blur_median();
//bin2.display("filtered3");

bin2.label_regions();
//bin2.display("label");

if(CEN_PERFORM){
QueryPerformanceCounter(&end2);
QueryPerformanceFrequency(&freq2);
QueryPerformanceCounter(&start3);
}
if (1) for (int i=0; i<bin2.dimx(); i++) {
for (int j=0; j<bin2.dimy(); j++) {
int k = bin2(i, j)-1;
if (k < 0) continue;
if (k == 0)
double fdfdf = 234.;
pixel.x=i; pixel.y=j; pixel.val = img(i,j);
if(stars[k].area<MAX_STAR_SIZE && k < MAX_STAR_NUM)
stars[k].pixels[stars[k].area++] = pixel;
if(k+1 > starNum)
starNum=k+1;
}
}
printf("\n current starnum is %d, thres was %d\n", starNum, thres);

if(CEN_PERFORM){
QueryPerformanceCounter(&end3);
QueryPerformanceFrequency(&freq3);
QueryPerformanceCounter(&start4);
}

//compute star centroid and brightness
if(1) for (int k=0; k<starNum; k++) {
xf = 0;
yf = 0;
brightness = 0;
```

```
for (int i=0; i<stars[k].area; i++) {
xf += (stars[k].pixels[i].x*stars[k].pixels[i].val);
yf += (stars[k].pixels[i].y*stars[k].pixels[i].val);
brightness += stars[k].pixels[i].val;
}
stars[k].cent.x = xf/brightness;
stars[k].cent.y = yf/brightness;
stars[k].brightness = brightness/stars[k].area;
}

cnt = 0;
for (int k=0; k<starNum; k++) {
tmp = cnt;
for (int i=cnt; i<starNum; i++) {
if (stars[i].area > stars[tmp].area) {
tmp = i;
}
}
tmpStar = stars[cnt];
stars[cnt] = stars[tmp];
stars[tmp] = tmpStar;
cnt++;
}

if(0) for (int k=0; k<starNum; k++) {
printf("star %d: (%f, %f), area %d, bright %f\n", k, stars[k].cent.x, stars[k].cent.y, stars[k].area, stars[k].brightness);
}

*numStar = starNum;
memcpy(&starCent[0], &stars[0], (starNum * sizeof(STAR)));
numrepeat++;
if(abs(WANTEDSTARNUM-starNum)>NUMROOM){
if(starNum>WANTEDSTARNUM)
thresmax=thres; // too many stars
else
thresmin=thres; // too few, raise the minimum
}
if(CEN_PERFORM){
QueryPerformanceCounter(&end4);
QueryPerformanceFrequency(&freq4);
double deltat = (double)(end.QuadPart - start.QuadPart)/(double)freq.QuadPart;
printf("\n centroid timer1 time=%f \n", deltat);
double deltat2 = (double)(end2.QuadPart - start2.QuadPart)/(double)freq2.QuadPart;
printf("\n centroid timer2 time=%f \n", deltat2);
double deltat3 = (double)(end3.QuadPart - start3.QuadPart)/(double)freq3.QuadPart;
printf("\n centroid timer3 time=%f \n", deltat3);
double deltat4 = (double)(end4.QuadPart - start4.QuadPart)/(double)freq4.QuadPart;
printf("\n centroid timer4 time=%f \n", deltat4);
}
} while(abs(WANTEDSTARNUM-starNum)>NUMROOM && numrepeat<MAXREPEAT);

return 0;
}

int main() {
//test (centroids)
```

```
int num;
int num2;
static STAR centroids[MAX_STAR_NUM];
static STAR centroids2[MAX_STAR_NUM];

LARGE_INTEGER start;
QueryPerformanceCounter(&start);

printf("centroids IMAGE 1 \n");
centroid("hol1a.bmp", &num, centroids);
printf("centroids IMAGE 2 \n");
centroid("hol1b.bmp", &num2, centroids2);

LARGE_INTEGER end, freq;
QueryPerformanceCounter(&end);
QueryPerformanceFrequency(&freq);
LARGE_INTEGER start2;
QueryPerformanceCounter(&start2);


/***
Reference Star Section
***/
//making fake imput to play with
ifstream fin ("hol1.txt");
int allstarnum=-1;
fin >> allstarnum;
StarArray RefStars(allstarnum); //bright
StarArray dimRefStars(allstarnum);
StarArray allRefStars(allstarnum);
for(int i=0; i<allstarnum; i++){
fin >> allRefStars[i].RA;
fin >> allRefStars[i].DEC;
fin >> allRefStars[i].pmRA;
fin >> allRefStars[i].pmDEC;
fin >> allRefStars[i].vMAG;
allRefStars[i].checknum=i;
allRefStars[i].alivebit=1;
fin >> allRefStars[i].catalogtype;
}

int brightstarnum=0;
int dimstarnum=0;
for (int i=0; i< allstarnum; i++){
if(allRefStars[i].catalogtype==0){
RefStars[brightstarnum]=allRefStars[i];
brightstarnum++;
}
else if(allRefStars[i].catalogtype==1){
dimRefStars[dimstarnum]=allRefStars[i];
dimstarnum++;
}
}

if(brightstarnum>NUMREFSTARS){
//pick the top NUMREFSTARS brightest stars from the bright star catalog (0)
```

```
double sortit0[MAX_STAR_NUM];
int numbright=0;
for(int i=0; i<brightstarnum; i++){
if(RefStars[i].catalogtype==0){
sortit0[i]=RefStars[i].vMAG;
numbright++;
}
}
qsort((void*) sortit0, numbright, sizeof(double), &cmp);
double minmag=sortit0[brightstarnum-NUMREFSTARS];
for(int i=0; i<brightstarnum; i++){
if(RefStars[i].vMAG<minmag && RefStars[i].catalogtype==0)
RefStars[i].alivebit=0;
}
}

if(dimstarnum>NUMREFSTARS){
//pick the middle NUMREFSTARS dim stars from the dim star catalog (1)
double sortit0[MAX_STAR_NUM];
int numbright=0;
for(int i=0; i<dimstarnum; i++){
if(dimRefStars[i].catalogtype==1){
sortit0[i]=dimRefStars[i].vMAG;
numbright++;
}
}
qsort((void*) sortit0, numbright, sizeof(double), &cmp);
int dimstarnum_min = int(dimstarnum/3);
int dimstarnum_max = int(2*dimstarnum/3);
double maxmag=sortit0[dimstarnum_max];
double minmag=sortit0[dimstarnum_min];
for(int i=0; i<dimstarnum; i++){
if(dimRefStars[i].vMAG>maxmag || dimRefStars[i].vMAG<minmag && dimRefStars[i].catalogtype==1)
dimRefStars[i].alivebit=0;
}
}

/***
Image Star Section
***/
StarArray ImgStarsA(num);
StarArray ImgStarsB(num2);
StarArray ImgStarsDim(num2);
int starnum2a=0;
int starnum2b=0;
int starnumdim=0;
if(TESTDATAMODE){
starnum2a=brightstarnum; //only uses one picture
for(int i=0; i<brightstarnum; i++){ //faking data
double theta=180*pi/180; //degrees rotated clockwise of data images from ref imgs
int j=i;
double introducedscale=1;
double offset1=.42;
double offset2=.36;
ImgStarsA[i].pmDEC=-1;
ImgStarsA[i].pmRA=-1;
```

```
ImgStarsA[i].vMAG=RefStars[j].vMAG*1.3;
ImgStarsA[i].RA= ( cos(theta)*(RefStars[j].RA+offset1)-sin(theta)*(RefStars[j].DEC+offset2))*introducedscale;
//<RA,Dec>
ImgStarsA[i].DEC=( sin(theta)*(RefStars[j].RA+offset1)+cos(theta)*(RefStars[j].DEC+offset2))*introducedscale;
ImgStarsA[i].checknum=RefStars[j].checknum;
ImgStarsA[i].alivebit=1;
}
}
else{
double maxbrightA=1000;
double maxbrightB=1000;
double maxbrightdim=1000;
if(num>NUMIMGSTARS){
double sortbrightnessA[MAXSTARSIMG];
for(int i=0; i<num; i++){
sortbrightnessA[i]=centroids[i].brightness;
}
qsort((void*) sortbrightnessA, num, sizeof(double), &cmp);
maxbrightA=sortbrightnessA[NUMIMGSTARS]; //Hard coded assertion that we deal with NUMIMGSTARS image
stars
}
for(int i=0; i<num; i++){
if(centroids[i].brightness<maxbrightA){
ImgStarsA[starnum2a].vMAG=centroids[i].brightness;
ImgStarsA[starnum2a].RA= centroids[i].cent.x; //<RA,Dec>
ImgStarsA[starnum2a].DEC=centroids[i].cent.y;
ImgStarsA[starnum2a].pmDEC=-1;
ImgStarsA[starnum2a].pmRA=-1;
ImgStarsA[starnum2a].checknum=-1;
ImgStarsA[starnum2a].alivebit=1;
starnum2a++;
}
}
if(num2>NUMIMGSTARS){
double sortbrightnessB[MAXSTARSIMG];
for(int i=0; i<num2; i++){
sortbrightnessB[i]=centroids2[i].brightness;
}
qsort((void*) sortbrightnessB, num2, sizeof(double), &cmp);
maxbrightB=sortbrightnessB[NUMIMGSTARS]; //Hard coded assertion that we deal with NUMIMGSTARS image
stars
maxbrightdim=sortbrightnessB[num2-NUMIMGSTARS]; //Hard coded assertion that we deal with
NUMIMGSTARS image stars
}
for(int i=0; i<num2; i++){
if(centroids2[i].brightness<maxbrightB){
ImgStarsB[starnum2b].vMAG=centroids2[i].brightness;
ImgStarsB[starnum2b].RA= centroids2[i].cent.x; //<RA,Dec>
ImgStarsB[starnum2b].DEC=centroids2[i].cent.y;
ImgStarsB[starnum2b].pmDEC=-1;
ImgStarsB[starnum2b].pmRA=-1;
ImgStarsB[starnum2b].checknum=-1;
ImgStarsB[starnum2b].alivebit=1;
starnum2b++;
}
if(centroids2[i].brightness>=maxbrightdim){
```

```
ImgStarsDim[starnumdim].vMAG=centroids2[i].brightness;
ImgStarsDim[starnumdim].RA= centroids2[i].cent.x; //<RA,Dec>
ImgStarsDim[starnumdim].DEC=centroids2[i].cent.y;
ImgStarsDim[starnumdim].pmDEC=-1;
ImgStarsDim[starnumdim].pmRA=-1;
ImgStarsDim[starnumdim].checknum=-1;
ImgStarsDim[starnumdim].alivebit=1;
starnumdim++;
}
}
}

//making triangles for image and reference stars

//for(int i=0; i<starnum2b; i++){
// printf("%d,%f,%f\n",i, ImgStarsB[i].RA,ImgStarsB[i].DEC);
//}
//for(int i=0; i<starnum2a; i++){
// printf("%d,%f,%f\n",i, ImgStarsA[i].RA,ImgStarsA[i].DEC);
//}


double scaleII=0, rotationII=0, translationxII=0, translationyII=0;
FindTriangle(ImgStarsB, starnum2b, ImgStarsA, starnum2a, 0, &scaleII, &rotationII, &translationxII,
&translationyII);
printf("**********\nImageA->ImageB::Scale=%f,Angle=%f,trans=<%f,%f>\n**********\n", scaleII, rotationII,
translationxII, translationyII);

double scaleIR=0, rotationIR=0, translationxIR=0, translationyIR=0;
FindTriangle(ImgStarsB, starnum2b, RefStars, brightstarnum, 0, &scaleIR, &rotationIR, &translationxIR,
&translationyIR, 0, 0, 0, 0, 183.07, 3.26, 1500);//38.39, 5.1, 2200.);
printf("**********\nImageB->Reference::Scale=%f,Angle=%f,trans=<%f,%f>\n**********\n ", scaleIR,
rotationIR, translationxIR, translationyIR);

StarArray ImgStarsC(starnum2b); //ImgStarsB, but converted into the A frame.
for(int i=0; i<starnum2b; i++){
ImgStarsC[i].vMAG=ImgStarsB[i].vMAG;
ImgStarsC[i].pmDEC=ImgStarsB[i].pmDEC;
ImgStarsC[i].pmRA=ImgStarsB[i].pmRA;
ImgStarsC[i].checknum=ImgStarsB[i].checknum;
ImgStarsC[i].alivebit=ImgStarsB[i].alivebit;
double answerx=0; double answery=0;
XYtoRADEC(ImgStarsB[i].RA, ImgStarsB[i].DEC, scaleII, rotationII, translationxII, translationyII, &answerx,
&answery);
ImgStarsC[i].RA= answerx; //<RA,Dec>
ImgStarsC[i].DEC= answery;
}

for(int i=0; i<starnum2b; i++){
double Pdistance=100;
for(int j=0; j<num; j++){
double Ndistance=sqrt((ImgStarsC[i].RA-centroids[j].cent.x)*(ImgStarsC[i].RA-
centroids[j].cent.x)+(ImgStarsC[i].DEC-centroids[j].cent.y)*(ImgStarsC[i].DEC-centroids[j].cent.y));
if(Ndistance<Pdistance)
Pdistance=Ndistance;
}
```

```
if(Pdistance>3){ //centroids more than 3 pixels apart=ASTEROID
double answerx=0; double answery=0;
XYtoRADEC(ImgStarsC[i].RA, ImgStarsC[i].DEC, scaleIR, rotationIR, translationxIR, translationyIR, &answerx,
&answery);
double currdist=100;
for(int j=0; j<dimstarnum; j++){
double Adistance=sqrt((answerx-dimRefStars[j].RA)*(answerx-dimRefStars[j].RA)+(answery-
dimRefStars[j].DEC)*(answery-dimRefStars[j].DEC));
if(Adistance<currdist)
currdist=Adistance;
}
if(currdist<.03)
printf("Possible Asteroid at XY(%f,%f) RADEC(%f,%f)\n",ImgStarsC[i].RA, ImgStarsC[i].DEC, answerx,
answery);
}
}

for (int k=0; k<starnum2b; k++) {
double answerx=0; double answery=0;
XYtoRADEC(ImgStarsC[k].RA, ImgStarsC[k].DEC, scaleIR, rotationIR, translationxIR, translationyIR, &answerx,
&answery);
ImgStarsC[k].RA= answerx; //<RA,Dec>
ImgStarsC[k].DEC= answery;
printf("star %d: %f %f %f %f\n", k, centroids2[k].cent.x, centroids2[k].cent.y, ImgStarsC[k].RA,
ImgStarsC[k].DEC);
}

LARGE_INTEGER end2, freq2;
QueryPerformanceCounter(&end2);
QueryPerformanceFrequency(&freq2);
double deltat = (double)(end.QuadPart - start.QuadPart)/(double)freq.QuadPart;
printf("\n centroid body timer time=%f \n", deltat);
double deltat2 = (double)(end2.QuadPart - start2.QuadPart)/(double)freq2.QuadPart;
printf("\n program body timer time=%f \n", deltat2);
printf("ending program\n");
return 0;

}
```

12.5 Orbital Determination Guide – Python and C

This code uses the Method of Gauss for a two-body program to calculate the orbital elements for every subset of three observations. An initial program was written in Python, but then converted into the C language for compatibility with the GPU. The python version of the code can be found below and the C version is part of the GPU code. The basic code structure is as follows.

First, we read in a file that contains the time, RA, and Dec for each observation. Then, the function `orbits` is called up which actually calculates the orbital elements of an orbit based on any three observations. However, to solve for the orbital elements, it is necessary to define the Earth-Sun vector at the times of observations. Therefore, inside the `orbits` function, there is an R-vector generator, or the Earth-Sun vector. This section essentially takes the known orbital

elements of the Earth, and plugs them into an ephemeris generator. From the orbital elements, the function is able to generate the vector relating the positions of the Earth and the Sun at any time. Using this vector, we can solve the fundamental vector triangle problem and determine the six orbital elements for an asteroid based on three observations. We can then loop the `orbits` function for every possible subset of three observations. From the list of all orbital element sets generated, we can go through and determine which orbital element sets are reasonable. We then cross-reference these orbital elements with the JPL-Horizons online asteroid database.

However, the generated orbital elements are not very accurate. In particular, the R-vector generator is not accurate due to the fact that more than two bodies are involved. There are many perturbations in the orbit of the Earth itself, and while these may not make a big initial difference, they do affect the final answer a great deal. Therefore, we looked elsewhere to find an n-body simulator, in which gravitational forces from all objects including planets, moons, and asteroids are accounted for.

12.6 Ephemeris Generator, Zeno - C

We found a site http://home.att.net/~srschmitt/planetorbits.html that offers a free ephemeris generator written in Zeno. Although we had an ephemeris generator written in Python, it did not account for variations in the orbital elements of the planets, which is crucial in generating an accurate Earth-Sun vector. We have since converted the program into the C language and assimilated it into our orbit determination code. A brief description of this process is as follows:

Given the time, it is possible to calculate the subtle changes in the orbital elements of the Earth. Using these modified orbital elements, we can then calculate the heliocentric radius of the Earth, based on the semi-major axis and eccentricity. We then must convert these coordinates into geocentric coordinates of the Sun with respect to the Earth. This will allow us to calculate the vector that relates the position of the Earth to the Sun at any time.

12.7 Python Orbit Determination Code

```
from __future__ import division
from visual import*
from visual.graph import*
from math import*

###############################################################################
#CONSTANTS
###############################################################################
k=.01720209895
cl=173.1446
mew=1
pi=4*arctan(1)
epsilon = 23.45/180*pi
```

```
###############################################################################
#FUNCTIONS
###############################################################################
def convertdms2d(x,y,z):
   degreeonly=x + y/60. + z/3600.
   return degreeonly

def converthms2d(x,y,z):
   degreeonly=(x + y/60. + z/3600.)*15.
   return degreeonly

def convertd2rad(deg):
   return deg*pi/180

def r2d (radian):
   return radian*180/pi

def equation(a,b,c,r2):
   return r2**8+a*r2**6+b*r2**3+c

def derv (a,b,c,r2):
   return 8.*r2**7.+6.*a*r2**5.+3.*b*r2**2.

def newson (a, b, c, userguess):
   slope=derv (a,b,c,userguess)
   yintercept=equation(a,b,c,userguess)-(slope)*userguess
   xintercept=-1*yintercept/slope
   return xintercept

def findangle(sin,cos):
   arcsin=r2d(asin(sin))
   arccos=r2d(acos(cos))
#if there the cos and sin input'ed don't match up
# if it is in the first quad the arccos and arcsin
#are positive and between 0 and 90
   if arcsin>0 and arccos < 90:
      return arcsin
# if arccos is more than 90 and less than 180 and arcsin is
#between 0 and 90, it is in the 2nd quad
   if arcsin>0 and arccos > 90:
      return arccos
#if arccos is between 0 and 90 and arcsin is in the 4th quad,
#it is in the 4th quad
   if arcsin<0 and arccos < 90:
      return arcsin+360
#if arccos is in 2nd quad and arcsin is in 4th quad, angle is in 3rd quad
   if arcsin<0 and arccos > 90:
      return 180+(180-arccos)
#for the 0, 90, 180, and 270
#0
   if arcsin==0 and arccos==0:
      return arcsin
#90
   if arcsin==90 and arccos==180:
      return arccos
#180
```

```
    if arcsin==0 and arccos==180:
        return arccos
#270
    if arcsin==-90 and arccos==0:
        return arcsin

def orbitalvector(a,e,E):
    x=a*cos(E)-a*e
    y=a*sqrt(1-e**2)*sin(E)
    z=0
    orb_orbital=[x,y,z]
    return orb_orbital

def rtoecliptic(x, y, z, omega, i, omegacap):
    x1=x*cos(omega)-y*sin(omega)
    y1=x*sin(omega)+y*cos(omega)
    z1=z
    x2=x1
    y2=y1*cos(i)-z1*sin(i)
    z2=y1*sin(i)+z1*cos(i)
    xec=x2*cos(omegacap)-y2*sin(omegacap)
    yec=x2*sin(omegacap)+y2*cos(omegacap)
    zec=z2
    orb_eccliptic=[xec,yec,zec]
    return orb_eccliptic

def ecliptictoequitorial(xec, yec, zec, epsilon):
    xeq=xec
    yeq=yec*cos(epsilon)-zec*sin(epsilon)
    zeq=yec*sin(epsilon)+zec*cos(epsilon)
    orb_equitorial=[xeq, yeq, zeq]
    return orb_equitorial

def findE(M, e):
    EGuess=M
    MGuess=EGuess-e*sin(EGuess)
    while abs(M-MGuess)>.000000000000000001:
        MGuess=EGuess-e*sin(EGuess)
        E=(M-MGuess)/(1-e*cos(EGuess))+EGuess
        EGuess=E
    return E
###############################################################################
#ORBIT DETERMINATION FUNCTION
###############################################################################
def orbits(t1, t2, t3, ra1, ra2, ra3, dec1, dec2, dec3):
#Calculate L Vectors
    L1=vector((cos(dec1)*cos(ra1)), cos(dec1)*sin(ra1),sin(dec1))
    L2=vector((cos(dec2)*cos(ra2)), cos(dec2)*sin(ra2),sin(dec2))
    L3=vector((cos(dec3)*cos(ra3)), cos(dec3)*sin(ra3),sin(dec3))

#Calculate proper time?
    tal1=k*(t1-t2)
    tal2=0.
    tal3=k*(t3-t2)

#Calculate R Vectors
```

```
#Earth Orbital Elements
   a=1.000732110928368E+00
   e=1.599910197101524E-02
   i=1.509328650209302E-03*pi/180
   omegacap=1.242780078662596E+02*pi/180
   omegasmall=3.392836473925709E+02*pi/180
   M2=1.784467221663580E+02*pi/180
   tmid=2453555.50000
   teph1=t1
   teph2=t2
   teph3=t3

   ###############################################################################
   #GENERATE R VECTORS
   ###############################################################################
   n=k*sqrt(mew/a**3)
   Meph1=n*(teph1-tmid)+M2
   Meph2=n*(teph2-tmid)+M2
   Meph3=n*(teph3-tmid)+M2

   E1=findE(Meph1, e)
   E2=findE(Meph2, e)
   E3=findE(Meph3, e)

   r_orb1=orbitalvector(a,e,E1)
   r_orb2=orbitalvector(a,e,E2)
   r_orb3=orbitalvector(a,e,E3)

   r_ec1=rtoecliptic(r_orb1[0], r_orb1[1], r_orb1[2], omegasmall, i, omegacap)
   r_ec2=rtoecliptic(r_orb2[0], r_orb2[1], r_orb2[2], omegasmall, i, omegacap)
   r_ec3=rtoecliptic(r_orb3[0], r_orb3[1], r_orb3[2], omegasmall, i, omegacap)

   R1=ecliptictoequitorial(r_ec1[0], r_ec1[1], r_ec1[2], epsilon)
   R2=ecliptictoequitorial(r_ec2[0], r_ec2[1], r_ec2[2], epsilon)
   R3=ecliptictoequitorial(r_ec3[0], r_ec3[1], r_ec3[2], epsilon)
   R1=(-R1[0],-R1[1],-R1[2])
   R2=(-R2[0],-R2[1],-R2[2])
   R3=(-R3[0],-R3[1],-R3[2])

   #R1=(9.350118648932805E-01, -3.009490457932475E-01, -1.304759343912905E-01)
   #R2=(9.919360806588228E-01, -6.891858264346408E-02, -2.987856846017327E-02)
   #R3=(9.071410046558456E-01, 3.932587816205667E-01, 1.704880435424688E-01)

   print "R1=", R1
   print "R2=", R2
   print "R3=", R3

#Define the D values
   d0=dot(L3,(cross(L1,L2)))
   d11=dot(L3,(cross(R1,L2)))
   d12=dot(L3,(cross(R2,L2)))
   d13=dot(L3,(cross(R3,L2)))
   d21=dot(L3,(cross(L1,R1)))
   d22=dot(L3,(cross(L1,R2)))
   d23=dot(L3,(cross(L1,R3)))
   d31=dot(L1,(cross(L2,R1)))
```

```
    d32=dot(L1,(cross(L2,R2)))
    d33=dot(L1,(cross(L2,R3)))

#Do calculation stuff
    dtal=tal3-tal1
    A1=tal3/dtal
    A3=-tal1/dtal
    B1=A1/6*(dtal**2-tal3**2)
    B3=A3/6*(dtal**2-tal1**2)

#Evaluate A,B,E,F
    A=-(A1*d21-d22+A3*d23)/d0
    B=-(B1*d21+B3*d23)/d0
    E=-2*(dot(L2,R2))
    F=dot(R2,R2)

#Evaluate a,b,c coefficents
    a=-(A**2+A*E+F)
    b=-mew*(2*A*B+B*E)
    c=-mew**2*B**2

#Newton's Method Thing
    guess = 1.
    answer=newson(a, b, c, guess)
#If the user actually guesses the zero
    if equation(a,b,c,guess)==0:
        print str(firstguess) +"good guess"
#Continue executing Newton's Method
    else:
        while abs((newson(a, b, c, answer)-answer))>=1*10**-8:
            answer=newson(a,b,c,answer)
    r2=answer

#Define approx f, g values
    u2=mew/r2**3
    f1=1-(u2/2)*tal1**2
    f3=1-(u2/2)*tal3**2
    g1=tal1-(u2/6)*(tal1**3)
    g3=tal3-(u2/6)*(tal3**3)

#Get the coefficents
    C1=g3/(f1*g3-f3*g1)
    C3=-g1/(f1*g3-f3*g1)
    C2=-1
#Find the P's
    P1=(C1*d11+C2*d12+C3*d13)/(C1*d0)
    P2=(C1*d21+C2*d22+C3*d23)/(C2*d0)
    P3=(C1*d31+C2*d32+C3*d33)/(C3*d0)
#Finding rvec1, rvec2, rvec3
    rvec1=P1*L1-R1
    rvec2=P2*L2-R2
    rvec3=P3*L3-R3
#Evaluate d coefficients
    d1=f3/(f3*g1-f1*g3)
    d3=-f1/(f3*g1-f1*g3)
#Find derv of rvec2
```

```
   dervrvec2=d1*rvec1+d3*rvec3
#Loop
   oldP2=P2+10
   while abs(oldP2-P2)>.00000001:
#16.1 correction for light time travel
      oldP2=P2
      t1c=t1-P1/173.1446
      t2c=t2-P2/173.1446
      t3c=t3-P3/173.1446
      tal1c=k*(t1c-t2c)
      tal3c=k*(t3c-t2c)
      dtal=tal3c-tal1c
   #16.2 define again the f and g
      zeta2=dot(rvec2,dervrvec2)/2
      u2=mew/(mag(rvec2)**3)
      Q2=dot(dervrvec2,dervrvec2)/(mag(rvec2)**2)-u2
      f1=1-(u2/2)*(tal1c**2)+u2*zeta2/2*(tal1c**3)+(1/24)*(3*u2*Q2-15*u2*zeta2**2+u2**2)*(tal1c)**4
      f3=1-(u2/2)*(tal3c**2)+u2*zeta2/2*(tal3c**3)+(1/24)*(3*u2*Q2-15*u2*zeta2**2+u2**2)*(tal3c)**4
      g1=tal1c-(u2/6)*(tal1c**3)+u2*zeta2/4*tal1c**4
      g3=tal3c-(u2/6)*(tal3c**3)+u2*zeta2/4*tal3c**4
   #16.3 Redo of all steps 11-15
      #11 get the coefficents
      C1=g3/(f1*g3-f3*g1)
      C3=-g1/(f1*g3-f3*g1)
      C2=-1
   #12/16 find the P's
      P1=(C1*d11+C2*d12+C3*d13)/(C1*d0)
      P2=(C1*d21+C2*d22+C3*d23)/(C2*d0)
      P3=(C1*d31+C2*d32+C3*d33)/(C3*d0)
   #13 Finding rvec1, rvec2, rvec3
      rvec1=P1*L1-R1
      rvec2=P2*L2-R2
      rvec3=P3*L3-R3
   #14 evaluate d coefficients
      d1=f3/(f3*g1-f1*g3)
      d3=-f1/(f3*g1-f1*g3)
   #15 find derv of rvec2
      dervrvec2=d1*rvec1+d3*rvec3

#Convert rvec2 and dervrvec2 to ecliptic
   eclip=23.45*pi/180
   rvec2ec=vector(rvec2.x,rvec2.y*cos(eclip)+rvec2.z*sin(eclip),-rvec2.y*sin(eclip)+rvec2.z*cos(eclip))
   dervrvec2ec=vector(dervrvec2.x,dervrvec2.y*cos(eclip)+dervrvec2.z*sin(eclip),-
dervrvec2.y*sin(eclip)+dervrvec2.z*cos(eclip))
#18find h##########################################################################################
   h=cross(rvec2ec,dervrvec2ec)
#19 Find
a##########################################################################################
   a=((2/mag(rvec2ec)-dot(dervrvec2ec,dervrvec2ec)/mew))**-1
   print "the value of a is " + str(a)
#20 the value of
e##########################################################################################
#
   ecc=sqrt(1-dot(h,h)/(mew*a))
   print "the value of e is " + str(ecc)
#find
```

```
i##############################################################################################
##############Put ur text here
   i=acos(h.z/mag(h))
   print "the value of i is " + str(r2d(i))
#22 finding
omega##########################################################################################
#######
   cosomegacap=-h.y/(mag(h)*sin(i))
   sinomegacap=h.x/(mag(h)*sin(i))
   cosangle=cosomegacap
   sinangle=sinomegacap
   omegacap=findangle(sinangle,cosangle)
   print "the value of capital omega is " + str(omegacap)
#Step 23 finding
U##############################################################################################
###########
   cosU=(rvec2ec.x*cos(omegacap*pi/180)+rvec2ec.y*sin(omegacap*pi/180))/mag(rvec2ec)
   sinU=(rvec2ec.z/(mag(rvec2ec)*sin(i)))
   U=findangle(sinU,cosU)
#Step 24 finding
nu#############################################################################################
#####################
   cosnu=(1/ecc)*(a*(1-ecc**2)/mag(rvec2ec)-1)
   sinnu=(a*(1-ecc**2)/ecc * dot(dervrvec2ec,rvec2ec)/(mag(h)*mag(rvec2ec)))
   nu = findangle(sinnu,cosnu)
#step 25
omegasmall#####################################################################################
#################
   omegasmall=U-nu
   if omegasmall<0:
      omegasmall=omegasmall+360
   print "the value of small omega " + str(omegasmall)
#Step 26 Get
esp#############################################################################################
#############
   if nu<0:
      nu=nu+360
   if nu>180:
      esp=360-(acos(1/ecc*(1-mag(rvec2ec)/a))*180/pi)
   if nu<180:
      esp= acos (1/ecc*(1-mag(rvec2ec)/a))*180/pi
#Step 27 Get M
##############################################################################################
############
   M=convertd2rad(esp)-ecc*sin(convertd2rad(esp))
   Mprint=M*180/pi
   print "the value of M is " + str(Mprint)
   orbelements = zeros(6, Float)
   orbelements[0] = a
   orbelements[1] = e
   orbelements[2] = i
   orbelements[3] = omegacap
   orbelements[4] = omegasmall
   orbelements[5] = Mprint
   return orbelements
```

```
################################################################################
#INPUT FROM FILES
################################################################################
file1=open("C:\Users/tony huang/Desktop/test.txt", "r")

#Set up arrays
time = zeros(3, Float)
RAhr = zeros(3, Float)
RAmin = zeros(3, Float)
RAsec = zeros(3, Float)
DECdeg = zeros(3, Float)
DECmin = zeros(3, Float)
DECsec = zeros(3, Float)
decdeg = zeros(3, Float)
radeg = zeros(3, Float)
ra = zeros(3, Float)
dec = zeros(3, Float)

for i in range(0,3):
   data=file1.readline()
   asteroid=data.split()
   time[i]=float(asteroid[0])
   RAhr[i]=float(asteroid[1])
   RAmin[i]=float(asteroid[2])
   RAsec[i]=float(asteroid[3])
   DECdeg[i]=float(asteroid[4])
   DECmin[i]=float(asteroid[5])
   DECsec[i]=float(asteroid[6])
file1.close()

################################################################################
#CONVERTING EVERYTHING TO RADIANS
################################################################################
for i in range(0,3):
   decdeg[i]=convertdms2d(DECdeg[i],DECmin[i],DECsec[i])
   radeg[i]=converthms2d(RAhr[i],RAmin[i],RAsec[i])
   dec[i]=convertd2rad(decdeg[i])
   ra[i]=convertd2rad(radeg[i])

################################################################################
#RUNNING EVERY SUBSET OF THREE OBSERVATIONS
################################################################################
elements = orbits(time[0], time[1], time[2], ra[0], ra[1], ra[2], dec[0], dec[1], dec[2])
```

12.8 GPU Orbit Determination Guide- C

The GPU program we used was a completely modified example program that originally multiplied matrices. None of the original functionality is still present, but modifying an existing program allowed for more of the CUDA compile setup to be present.

Main.cu contains the main function call, orbitfind, included in angle brackets. This file is run on the CPU, allowing it to initialize the GPU. Orbitfind is in the kernel.cu file, which is the GPU's processor initializer. It calls GPU_OD_Calc, included in bracketfunc.cpp. Each processor runs this funciton seperately. Some loops have been put into this function as well as main.cu to

test for whether or not the program is pipelining.

12.9 GPU Orbit Determination Code

12.9.1 Main.cu

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <Windows.h>

// includes, project
#include <cutil.h>

// includes, kernels
#include <kernel.cu>

////////////////////////////////////////////////////////////////////////////////
// declaration, forward
void runTest(int argc, char** argv);

extern "C" void readinfile(double *,double *, double *);

////////////////////////////////////////////////////////////////////////////////
// Program main
////////////////////////////////////////////////////////////////////////////////
int
main(int argc, char** argv)
{
runTest(argc, argv);

CUT_EXIT(argc, argv);
}

////////////////////////////////////////////////////////////////////////////////
//! Run a simple test for CUDA
////////////////////////////////////////////////////////////////////////////////
void
runTest(int argc, char** argv)
{
CUT_DEVICE_INIT();
double times[NUMOBS]; double ras[NUMOBS]; double decs[NUMOBS];

readinfile(times, ras, decs);
int maxL=0;
for(int i=0; i<NUMPROCESSORS; i++){
int stride = 1<<i;
if (stride>=NUMPROCESSORS&& maxL==0){
maxL=i;
}
}
// allocate device memory
static allobs parms;
// array of data structures
```

```
for(int i=0; i<NUMOBS; i++){
parms.ob[i].JulianDate=times[i];
parms.ob[i].RA=ras[i];
parms.ob[i].DEC=decs[i];
}
allobs *d_contents;

printf("beginning program\n");
LARGE_INTEGER start;
QueryPerformanceCounter(&start);
CUDA_SAFE_CALL(cudaMalloc((void**) &d_contents, sizeof(allobs)));

// copy host memory to device
CUDA_SAFE_CALL(cudaMemcpy((char *)d_contents, (char *)&parms, sizeof(allobs),
cudaMemcpyHostToDevice) );

// create and start timer
unsigned int timer = 0;
CUT_SAFE_CALL(cutCreateTimer(&timer));
CUT_SAFE_CALL(cutStartTimer(timer));

// setup execution parameters
dim3 threads2(1, NUMPROCESSORS);
dim3 grid2(1, (NUMANSWER+7)/8*1000);
// execute the kernel
orbitfind<<< grid2, threads2 >>>(d_contents, maxL);
// check if kernel execution generated and error
CUT_CHECK_ERROR("Kernel execution failed");

// copy result from device to host
allobs answer;
CUDA_SAFE_CALL(cudaMemcpy((char *)&answer, (char *)d_contents, sizeof(allobs),
cudaMemcpyDeviceToHost) );
LARGE_INTEGER end, freq;
QueryPerformanceCounter(&end);
QueryPerformanceFrequency(&freq);
double deltat = (double)(end.QuadPart - start.QuadPart)/(double)freq.QuadPart;
printf("\n timer time=%f \n", deltat);
//printing bracket result
printf("\n Printing Results GPU \n");
for(int i=0; i<NUMANSWER; i++){
printf("%f %f %f %f %f %f \n", answer.answers[i].Asemimajor, answer.answers[i].Aecc,
answer.answers[i].Ainclination, answer.answers[i].Aomegacap, answer.answers[i].Aomegasmall,
answer.answers[i].AM);
}
printf("\n \n");

// stop and destroy timer
CUT_SAFE_CALL(cutStopTimer(timer));
//printf("Processing time: %f (ms) n", cutGetTimerValue(timer));
CUT_SAFE_CALL(cutDeleteTimer(timer));
getchar();
// clean up memory
CUDA_SAFE_CALL(cudaFree(d_contents));
}
```

12.9.2 Kernal.cu

```
#ifndef _MATRIXMUL_KERNEL_H_
#define _MATRIXMUL_KERNEL_H_
#include <stdio.h>
#include "matrixMul.h"
#include "bracketfunc.cpp"
__global__ void orbitfind(allobs* contents, int maxL) {
int index= blockIdx.y*NUMPROCESSORS+threadIdx.y;
GPU_OD_Calc(index, contents);
}
#endif
```

12.9.3 Bracketfunc.cpp

```
#define ASSERT(c)
#define pow2(x) ((x)*(x))
#define pow3(x) ((x)*(x)*(x))
#define pow4(x) ((x)*(x)*(x)*(x))

class tvec{
public:
float x;
float y;
float z;

tvec(float a, float b, float c){
x=a;
y=b;
z=c;
}
tvec() { x = 0.; y = 0.; z=0.; }
};
__device__ float findE(float M, float e){
float E=0;
float EGuess=M;
float MGuess=EGuess-e*sin(EGuess);
while (abs(M-MGuess)>.000000000000000001){
MGuess=EGuess-e*sin(EGuess);
E=(M-MGuess)/(1-e*cos(EGuess))+EGuess;
EGuess=E;
}
return E;
}
__device__ tvec orbitalvector(float a, float e, float E){
float x=a*cos(E)-a*e;
float y=a*sqrt(1-pow2(e))*sin(E);
float z=0;
tvec orb_orbital(x,y,z);
return orb_orbital;
}

__device__ tvec rtoecliptic(float x, float y, float z, float omega,float i,float omegacap){
float x1=x*cos(omega)-y*sin(omega);
```

```
float y1=x*sin(omega)+y*cos(omega);
float z1=z;
float x2=x1;
float y2=y1*cos(i)-z1*sin(i);
float z2=y1*sin(i)+z1*cos(i);
float xec=x2*cos(omegacap)-y2*sin(omegacap);
float yec=x2*sin(omegacap)+y2*cos(omegacap);
float zec=z2;
tvec orb_eccliptic(xec,yec,zec);
return orb_eccliptic;
}

__device__ tvec eclipticto equitorial(float xec, float yec, float zec, float epsilon){
float xeq=xec;
float yeq=yec*cos(epsilon)-zec*sin(epsilon);
float zeq=yec*sin(epsilon)+zec*cos(epsilon);
tvec orb_equitorial(xeq, yeq, zeq);
return orb_equitorial;
}
__device__ float dot(tvec a, tvec b){
float answer = a.x*b.x + a.y*b.y + a.z*b.z;
return answer;
}

__device__ tvec cross(tvec a, tvec b){
tvec answer(a.y*b.z - a.z*b.y, a.z*b.x - a.x*b.z, a.x*b.y - a.y*b.x);
return answer;
}

__device__ float mag(tvec a) {
return sqrtf(a.x*a.x + a.y*a.y + a.z*a.z);
}
__device__ float mag2(tvec a) {
return a.x*a.x + a.y*a.y + a.z*a.z;
}
__device__ float mag3(tvec a) {
float mag2 = a.x*a.x + a.y*a.y + a.z*a.z;
return mag2*sqrtf(mag2);
}
__device__ float equation(float a,float b,float c,float r2){
float r2sq = r2*r2;
float r2fourth = r2sq*r2sq;
return r2fourth*r2fourth + a*r2fourth*r2sq + b*r2*r2sq + c;
}
__device__ float derv (float a,float b,float c,float r2){
return 8*pow(r2,7)+6*a*pow(r2,5)+3*b*pow2(r2);
}
__device__ float newton (float a,float b,float c,float userguess){
float slope=derv (a,b,c,userguess);
float yintercept=equation(a,b,c,userguess)-(slope)*userguess;
float xintercept=-1*yintercept/slope;
return xintercept;
}
__device__ float convertd2rad(float deg){
return deg*3.14159265358979323846/180;
}
```

```
__device__ float mod2pi(double angle){
while (angle > (2*3.14159265358979323846))
angle = angle - (2*3.14159265358979323846);
while (angle < 0)
angle = angle + (2*3.14159265358979323846);
return angle;
}

__device__ float r2d (float radian){
return (float)(radian*180/3.14159265358979323846);
}

__device__ float findangle(float sinp, float cosp){
float arcsin=r2d(asin(sinp));
float arccos=r2d(acos(cosp));
////if there the cos and sin input'ed don't match up
//// if it is in the first quad the arccos and arcsin
////are positive and between 0 and 90
if (arcsin>0 && arccos < 90)
return arcsin;
//// if arccos is more than 90 and less than 180 and arcsin is
////between 0 and 90, it is in the 2nd quad
if (arcsin>0 && arccos > 90)
return arccos;
////if arccos is between 0 and 90 and arcsin is in the 4th quad,
////it is in the 4th quad
if (arcsin<0 && arccos < 90)
return arcsin+360;
////if arccos is in 2nd quad and arcsin is in 4th quad, angle is in 3rd quad
if (arcsin<0 && arccos > 90)
return 180+(180-arccos);
////for the 0, 90, 180, and 270
////0
if (arcsin==0 && arccos==0)
return arcsin;
////90
if (arcsin==90 && arccos==180)
return arccos;
////180
if (arcsin==0 && arccos==180)
return arccos;
////270
if (arcsin==-90 && arccos==0)
return arcsin;
return 0.;
}
__device__ void GPU_OD_Calc(int procnum, allobs *passedmem) {
while (procnum >= NUMANSWER) procnum -= NUMANSWER;
#define LOOPER 1
int looper = LOOPER;
labelxx: 0;
int k=0, l=0, m=0, runningindex=0;
while (1){
int gaplength=(NUMOBS-k-1)*(NUMOBS-k-2)/2;
if(procnum<(runningindex+gaplength))
```

```
break;
k++;
runningindex+=gaplength;
}
int rownumber=(1+(int)(sqrt((float)(1+8*(procnum-runningindex)))))/2;
l=k+1+procnum-runningindex-rownumber*(rownumber-1)/2;
m=k+1+rownumber;
//printf("procnum %d with obs %d, %d, %d\n", procnum, k, l,m);
float pi=(float)(3.14159265358979323846);
#if 1
#define xyz(n) __device__ __shared__ float w##n##x[NUMPROCESSORS];
xyz(1); xyz(2);
__device__ __shared__ float w3x[NUMPROCESSORS], w4x[NUMPROCESSORS], w5x[NUMPROCESSORS],
w6x[NUMPROCESSORS], w7x[NUMPROCESSORS], w8x[NUMPROCESSORS], w9x[NUMPROCESSORS],
w10x[NUMPROCESSORS], w11x[NUMPROCESSORS], w12x[NUMPROCESSORS],
w13x[NUMPROCESSORS], w14x[NUMPROCESSORS], w15x[NUMPROCESSORS],
w16x[NUMPROCESSORS], w17x[NUMPROCESSORS], w18x[NUMPROCESSORS],
w19x[NUMPROCESSORS], w20x[NUMPROCESSORS], w21x[NUMPROCESSORS],
w22x[NUMPROCESSORS], w23x[NUMPROCESSORS], w24x[NUMPROCESSORS],
w26x[NUMPROCESSORS], w27x[NUMPROCESSORS];
#define w1 w1x[procnum]
#define w2 w2x[procnum]
#define w3 w3x[procnum]
#define w4 w4x[procnum]
#define w5 w5x[procnum]
#define w6 w6x[procnum]
#define w7 w7x[procnum]
#define w8 w8x[procnum]
#define w9 w9x[procnum]
#define w10 w10x[procnum]
#define w11 w11x[procnum]
#define w12 w12x[procnum]
#define w13 w13x[procnum]
#define w14 w14x[procnum]
#define w15 w15x[procnum]
#define w16 w16x[procnum]
#define w17 w17x[procnum]
#define w18 w18x[procnum]
#define w19 w19x[procnum]
#define w20 w20x[procnum]
#define w21 w21x[procnum]
#define w22 w22x[procnum]
#define w23 w23x[procnum]
#define w24 w24x[procnum]
#define w26 w26x[procnum]
#define w27 w27x[procnum]
#else
float w4 = 0;
float w1=0, w2=0, w3=0, /*w4=0,*/ w5=0, w6=0, w7=0, w8=0, w9=0, w10=0, w11=0, w12=0, w13=0, w14=0,
w15=0, w16=0, w17=0, w18=0, w19=0, w20=0, w21=0, w22=0, w23=0, w24=0, w26=0, w27=0;
#endif
tvec v1, v2, v3, v4, v5, v6, v7, v8, v9, v10;
//STEP #4 - Calculate R Vectors
//////////////////////////
//EARTH ORBITAL ELEMENTS... have been incorporated so as not to take up space//
//////////////////////////
```

```
//teph1=t1, for 1 througth 3

//////////////////////
//GENERATE R VECTORS//
//////////////////////
#define R1 v1
#define R2 v2
#define R3 v3
#define n w1
n=(float)((.01720209895)*sqrt(1/pow3(1.000732110928368E+00)));
#define r_orb1 v1
#define r_orb2 v2
#define r_orb3 v3
r_orb1=orbitalvector((1.000732110928368E+00),(1.599910197101524E-02),findE(n*(passedmem-
>ob[k].JulianDate-(2453555.50000))+(1.784467221663580E+02*pi/180), (1.599910197101524E-02)));
r_orb2=orbitalvector((1.000732110928368E+00),(1.599910197101524E-02),findE(n*(passedmem-
>ob[l].JulianDate-(2453555.50000))+(1.784467221663580E+02*pi/180), (1.599910197101524E-02)));
r_orb3=orbitalvector((1.000732110928368E+00),(1.599910197101524E-02),findE(n*(passedmem-
>ob[m].JulianDate-(2453555.50000))+(1.784467221663580E+02*pi/180), (1.599910197101524E-02)));
#undef n
#define r_ec1 v4
#define r_ec2 v5
#define r_ec3 v6
r_ec1=rtoecliptic(r_orb1.x, r_orb1.y, r_orb1.z, (3.392836473925709E+02*pi/180), (1.509328650209302E-
03*pi/180), (1.242780078662596E+02*pi/180));
r_ec2=rtoecliptic(r_orb2.x, r_orb2.y, r_orb2.z, (3.392836473925709E+02*pi/180), (1.509328650209302E-
03*pi/180), (1.242780078662596E+02*pi/180));
r_ec3=rtoecliptic(r_orb3.x, r_orb3.y, r_orb3.z, (3.392836473925709E+02*pi/180), (1.509328650209302E-
03*pi/180), (1.242780078662596E+02*pi/180));
#undef r_orb1
#undef r_orb2
#undef r_orb3
R1=ecliptictoequitorial(r_ec1.x, r_ec1.y, r_ec1.z, (23.45/180*pi));
R2=ecliptictoequitorial(r_ec2.x, r_ec2.y, r_ec2.z, (23.45/180*pi));
R3=ecliptictoequitorial(r_ec3.x, r_ec3.y, r_ec3.z, (23.45/180*pi));
#undef r_ec1
#undef r_ec2
#undef r_ec3
R1.x = -R1.x;
R1.y = -R1.y;
R1.z = -R1.z;
R2.x = -R2.x;
R2.y = -R2.y;
R2.z = -R2.z;
R3.x = -R3.x;
R3.y = -R3.y;
R3.z = -R3.z;
//printf("R1=%f,%f,%f", R1.x, R1.y, R1.z);
//printf("\nR2=%f,%f,%f", R2.x, R2.y, R2.z);
//printf("\nR3=%f,%f,%f", R3.x, R3.y, R3.z);

//STEP #2 - Calculate L Vectors (moved)
#define L1 v4
#define L2 v5
#define L3 v6
```

```
L1=tvec((cos(passedmem->ob[k].DEC)*cos(passedmem->ob[k].RA)), cos(passedmem-
>ob[k].DEC)*sin(passedmem->ob[k].RA),sin(passedmem->ob[k].DEC));
L2=tvec((cos(passedmem->ob[l].DEC)*cos(passedmem->ob[l].RA)), cos(passedmem-
>ob[l].DEC)*sin(passedmem->ob[l].RA),sin(passedmem->ob[l].DEC));
L3=tvec((cos(passedmem->ob[m].DEC)*cos(passedmem->ob[m].RA)), cos(passedmem-
>ob[m].DEC)*sin(passedmem->ob[m].RA),sin(passedmem->ob[m].DEC));

//STEP #3 - Calculate proper time?
#define tal1 w1
#define tal3 w2
tal1=(.01720209895)*(passedmem->ob[k].JulianDate-passedmem->ob[l].JulianDate);
tal3=(.01720209895)*(passedmem->ob[m].JulianDate-passedmem->ob[l].JulianDate);
//STEP #5 - Define the D values
#define d0 w3
#define d11 w4
#define d12 w5
#define d13 w6
#define d21 w7
#define d22 w8
#define d23 w9
#define d31 w10
#define d32 w11
#define d33 w12
d0=dot(L3,(cross(L1,L2)));
d11=dot(L3,(cross(R1,L2)));
d12=dot(L3,(cross(R2,L2)));
d13=dot(L3,(cross(R3,L2)));
d21=dot(L3,(cross(L1,R1)));
d22=dot(L3,(cross(L1,R2)));
d23=dot(L3,(cross(L1,R3)));
d31=dot(L1,(cross(L2,R1)));
d32=dot(L1,(cross(L2,R2)));
d33=dot(L1,(cross(L2,R3)));

//STEP #6 - Calculate A1, B1, A2, B2
#define dtal w13
dtal=tal3-tal1;
//STEP #7 - Evaluate A,B,E,F
#define A w14
#define B w15
#define E w16
A=-((tal3/dtal)*d21-d22+(-tal1/dtal)*d23)/d0;
B=-(((tal3/dtal)/6*(pow2(dtal)-pow2(tal3)))*d21+((-tal1/dtal)/6*(pow2(dtal)-pow2(tal1)))*d23)/d0;
E=-2*(dot(L2,R2));

//STEP #8 - Evaluate a,b,c coefficents
#define a w17
#define b w18
#define c w19
a=-(pow2(A)+A*E+dot(R2,R2));
b=-1*(2*A*B+B*E);
c=-pow2(B);
#undef A
#undef B
#undef E
//STEP #9 - Newton's Method Thing
```

```
#define r2 w14
#define NEWTONBASE 1.39
r2=newton(a, b, c, NEWTONBASE);
//If the user actually guesses the zero
if (equation(a,b,c,NEWTONBASE)==0)
;//printf("%f good guess", getchar());
//Continue executing Newton's Method
else {
int lpc;
for (lpc = 0; lpc < 15; lpc++) {
if (abs((newton(a, b, c, r2)-r2))>=0.000001)
r2=newton(a,b,c,r2);
else
break;
}
passedmem->answers[procnum].Aomegasmall = lpc;
}

#undef answer
#undef a
#undef b
#undef c

//STEP #10 - Define approx f, g values
#define u2 w14
#define f1 w15
#define f3 w16
#define g1 w17
#define g3 w18
u2=1/pow3(r2);
f1=1-(u2/2)*pow2(tal1);
f3=1-(u2/2)*pow2(tal3);
g1=tal1-(u2/6)*pow3(tal1);
g3=tal3-(u2/6)*pow3(tal3);
#undef tal1
#undef tal3

//STEP #11 - Get the C-coefficents
//STEP #12 - Find the P's
#define P1 w1
#define P2 w2
#define P3 w19
P1=((g3/(f1*g3-f3*g1))*d11+(-1)*d12+(-g1/(f1*g3-f3*g1))*d13)/((g3/(f1*g3-f3*g1))*d0);;
P2=((g3/(f1*g3-f3*g1))*d21+(-1)*d22+(-g1/(f1*g3-f3*g1))*d23)/((-1)*d0);
P3=((g3/(f1*g3-f3*g1))*d31+(-1)*d32+(-g1/(f1*g3-f3*g1))*d33)/((-g1/(f1*g3-f3*g1))*d0);
//STEP #13 - Finding rvec1, rvec2, rvec3
#define rvec1 v7
#define rvec2 v8
#define rvec3 v9
rvec1.x=P1*L1.x-R1.x;
rvec1.y=P1*L1.y-R1.y;
rvec1.z=P1*L1.z-R1.z;
rvec2.x=P2*L2.x-R2.x;
rvec2.y=P2*L2.y-R2.y;
rvec2.z=P2*L2.z-R2.z;
rvec3.x=P3*L3.x-R3.x;
```

```
rvec3.y=P3*L3.y-R3.y;
rvec3.z=P3*L3.z-R3.z;
//STEP #14 - Evaluate d coefficients
#define d1 w20
#define d3 w21
d1=f3/(f3*g1-f1*g3);
d3=-f1/(f3*g1-f1*g3);
//STEP #15 - Find derv of rvec2
#define dervrvec2 v10
dervrvec2.x=d1*rvec1.x+d3*rvec3.x;
dervrvec2.y=d1*rvec1.y+d3*rvec3.y;
dervrvec2.z=d1*rvec1.z+d3*rvec3.z;
//STEP #16 - Loop to refine elements
#define oldP2 w22
oldP2=P2+10;
#define tal1c w23
#define tal3c w24
//#define dtal w25
#define zeta2 w26
#define Q2 w27
int loopcounter=0;
while (abs(oldP2-P2)>.00000000000000000001&& loopcounter<500){
//16.1 correction for light time travel
oldP2=P2;
tal1c=(.01720209895)*((passedmem->ob[k].JulianDate-P1/173.1446)-(passedmem->ob[l].JulianDate-
P2/173.1446));
tal3c=(.01720209895)*((passedmem->ob[m].JulianDate-P3/173.1446)-(passedmem->ob[l].JulianDate-
P2/173.1446));
dtal=tal3c-tal1c;
//16.2 define again the f and g
zeta2=dot(rvec2,dervrvec2)/2;
u2=1/(mag3(rvec2));
Q2=dot(dervrvec2,dervrvec2)/(mag2(rvec2))-u2;
f1=1-(u2/2)*pow2(tal1c)+u2*zeta2/2*pow3(tal1c)+(1/24)*(3*u2*Q2-15*u2*pow2(zeta2)+pow2(u2)*pow4(tal1c));
f3=1-(u2/2)*pow2(tal3c)+u2*zeta2/2*pow3(tal3c)+(1/24)*(3*u2*Q2-15*u2*pow2(zeta2)+pow2(u2)*pow4(tal3c));
g1=tal1c-(u2/6)*pow3(tal1c)+u2*zeta2/4*pow4(tal1c);
g3=tal3c-(u2/6)*pow3(tal3c)+u2*zeta2/4*pow4(tal3c);
//16.3 Redo of all steps 11-15
//check out why these need to be re-doubled
P1=((g3/(f1*g3-f3*g1))*d11+(-1)*d12+(-g1/(f1*g3-f3*g1))*d13)/((g3/(f1*g3-f3*g1))*d0);
P2=((g3/(f1*g3-f3*g1))*d21+(-1)*d22+(-g1/(f1*g3-f3*g1))*d23)/((-1)*d0);
P3=((g3/(f1*g3-f3*g1))*d31+(-1)*d32+(-g1/(f1*g3-f3*g1))*d33)/((-g1/(f1*g3-f3*g1))*d0);

rvec1.x=P1*L1.x-R1.x;
rvec1.y=P1*L1.y-R1.y;
rvec1.z=P1*L1.z-R1.z;
rvec2.x=P2*L2.x-R2.x;
rvec2.y=P2*L2.y-R2.y;
rvec2.z=P2*L2.z-R2.z;
rvec3.x=P3*L3.x-R3.x;
rvec3.y=P3*L3.y-R3.y;
rvec3.z=P3*L3.z-R3.z;

d1=f3/(f3*g1-f1*g3);
d3=-f1/(f3*g1-f1*g3);
```

```
dervrvec2.x=d1*rvec1.x+d3*rvec3.x;
dervrvec2.y=d1*rvec1.y+d3*rvec3.y;
dervrvec2.z=d1*rvec1.z+d3*rvec3.z;
loopcounter++;
}
#undef d0
#undef d11
#undef d12
#undef d13
#undef d21
#undef d22
#undef d23
#undef d31
#undef d32
#undef d33
#undef dtal
#undef R1
#undef R2
#undef R3
#undef L1
#undef L2
#undef L3
#undef rvec1
#undef rvec3
#undef u2
#undef f1
#undef f3
#undef g1
#undef g3
#undef P1
#undef P2
#undef P3
#undef d1
#undef d3
#undef oldP2
#undef tal1c
#undef tal3c
#undef dtal
#undef zeta2
#undef Q2

#define eclip w1
//STEP #17 - Convert rvec2 and dervrvec2 to ecliptic
//eclip=23.45*pi/180;
#define ceclip .917407699
#define seclip .397948631
#define rvec2ec v1
rvec2ec=tvec(rvec2.x,rvec2.y*ceclip+rvec2.z*seclip,-rvec2.y*seclip+rvec2.z*ceclip);

#undef rvec2
#define dervrvec2ec v2
dervrvec2ec=tvec(dervrvec2.x,dervrvec2.y*ceclip+dervrvec2.z*seclip,-dervrvec2.y*seclip+dervrvec2.z*ceclip);
#undef dervrvec2
//STEP #18 - Calculate h-vector
#define h v3
h=cross(rvec2ec,dervrvec2ec);
```

```
//STEP #19 - Calculate a(semi-major axis)
#define semimajor w2
semimajor=1/((2/mag(rvec2ec)-dot(dervrvec2ec,dervrvec2ec)/1));
//printf( "\nthe value of a is, %f ", semimajor);
//STEP #20 - Calculate e(eccentricity)
#define ecc w3
ecc=sqrt(1-dot(h,h)/(1*semimajor));
//printf( "\nthe value of e is, %f ", ecc);
//STEP #21 - Calculate procnum(inclination)
#define inclination w4
inclination=acos(h.z/mag(h));
//printf( "\nthe value of procnum is, %f ", r2d(inclination));
//STEP #22 - Calculate omega cap
#define cosomegacap w5
#define sinomegacap w6
#define cosangle w7
#define sinangle w8
#define omegacap w9
cosomegacap=-h.y/(mag(h)*sin(inclination));
sinomegacap=h.x/(mag(h)*sin(inclination));
cosangle=cosomegacap;
sinangle=sinomegacap;
omegacap=findangle(sinangle,cosangle);
//printf( "\nthe value of capital omega is, %f ", omegacap);
//STEP #23 - Find U
#define cosU w10
#define sinU w11
#define U w12
#define cosnu w13
#define sinnu w14
cosU=(rvec2ec.x*cos(omegacap*pi/180)+rvec2ec.y*sin(omegacap*pi/180))/mag(rvec2ec);
sinU=(rvec2ec.z/(mag(rvec2ec)*sin(inclination)));
U=findangle(sinU,cosU);
//STEP #24 - Find nu
cosnu=(1/ecc)*((semimajor*(1-pow2(ecc))/mag(rvec2ec))-1);
sinnu=(semimajor*(1-pow2(ecc))/ecc * dot(dervrvec2ec,rvec2ec)/(mag(h)*mag(rvec2ec)));
#undef dervrvec2ec
#undef h
#define nu w15
#define omegasmall w16
#define esp w17
nu = findangle(sinnu,cosnu);
//STEP #25 - Calculate omega small
omegasmall=U-nu;
esp=0;
if (omegasmall<0)
omegasmall=omegasmall+360;
//printf( "\nthe value of small omega, %f ", omegasmall);
//STEP #26 - Find esp
if (nu<0)
nu=nu+360;
if (nu>180)
esp=360-(acos(1/ecc*(1-mag(rvec2ec)/semimajor))*180/pi);
if (nu<180)
esp= acos (1/ecc*(1-mag(rvec2ec)/semimajor))*180/pi;
#undef rvec2ec
```

```
//STEP #27 - Calculate M(mean anomaly)
#define M w18
M=convertd2rad(esp)-ecc*sin(convertd2rad(esp));

passedmem->answers[procnum].AM = M*180/pi;
passedmem->answers[procnum].Asemimajor = semimajor;
if (--looper > 0) goto labelxx;
passedmem->answers[procnum].Aecc = ecc;
passedmem->answers[procnum].Ainclination = inclination;
passedmem->answers[procnum].Aomegacap = omegacap;
passedmem->answers[procnum].Aomegasmall = omegasmall;
passedmem->answers[procnum].Aomegasmall = procnum;
passedmem->answers[procnum].AM = M;
}
```