E90 PROJECT:
STEREO VISUAL ODOMETRY

By: Yavor Georgiev

Advisor: Bruce Maxwell

Date: Saturday, May 06, 2006

I. TABLE OF CONTENTS

II. LIST OF FIGURES

III. LIST OF TABLES

IV. ABSTRACT

This paper presents an implementation of a visual odometry system as described by David Nistér using Videre Design's STH-DCSG Firewire stereo video head, which runs on an AMD Athlon-based workstation with 1GB RAM. The system tracks the 3D position of a calibrated stereo head as it moves through the environment. Harris corner features are identified in the video stream, and are then matched and triangulated into 3D points. A RANSAC implementation with a quaternion motion estimation method as its hypothesis generator, is used for pose estimation. The system is able to estimate movement in the camera's x and y planes with a maximum error of 12cm for a duration of a minute or longer. A persistent drift in the z dimension (along the camera optical ray) prevents the extraction of useful movement data along that axis.

V. INTRODUCTION

Accurate localization and mapping is a problem of great significance in the mobile robotics community, and especially in the context of Urban Search and Rescue (USR) applications, where a teleoperated (or sometimes autonomous) agent is dispatched inside an unknown hazardous environment to collect sensory data on a time-critical task. Almost always the environments surveyed are unstructured and previously unknown, so creating accurate maps is essential should a human rescue effort be necessary.

There are a number of notable localization methods being used today in the robotics community, but visual odometry is the only approach that fit our performance needs and cost constraints. We considered sonar-based methods, but the results[1] obtained by other researchers were not as general as we needed. Laser range-finder solutions were definitely very well-suited to our task, but the cost of the equipment necessary was prohibitive. Visual odometry, as presented in the work[2] of David Nistér emerged as a good tradeoff between reliability, cost, and implementation complexity. The necessary stereo equipment had already been purchased by Prof. Maxwell using grant funds, and this implicit expense has been excluded from the scope of this project. Table 1 summarizes the advantages and drawbacks of each localization method.

Table 1. Comparison of localization methods

Method	Advantages	Drawbacks
DGPS	• Excellent resolution (1cm) • Ease of implementation • Moderate cost	• Doesn't work indoors
Sonar + IR	• Low cost	• Very hard to get robust performance • Multiple error modes
Laser range-finder	• Excellent resolution • Well-understood implementation	• High cost (~$5K)
Visual odometry	• Low cost (~$1K) • Potentially very accurate	• Lots of difficult implementation details • Only operates in good light conditions

VI. SETUP

6.1 SYSTEM COMPONENTS

The stereo camera used in this project was Videre Design's STH-DCSG/C Firewire head[3] (Figure 1), used in grayscale mode at a resolution of 320 x 240px. The system was run on a dual-processor AMD Athlon workstation with 1GB of RAM, running Debian Linux.

Figure 1. STH-DCSG/C stereo head

The Small Vision System (SVS)[4] C++ libsvs.so library[5] was used to transfer images from the camera, and to perform some triangulation calculations. To calibrate the camera we used the *smallvcal* utility[6], which is also part of SVS.

For fast and efficient implementations of eigenvalue calculation, singular-value decomposition, and some basic matrix algebra, the Newmat C++ Matrix Library ver. 10[7] was used.

In addition, the OpenCV[8] *goodFeaturesToTrack* function was used as an alternative implementation of Harris corner feature detector, and the *cvPOSIT* function was used to iteratively solve the 3-ponit pose estimation problem.

6.2 CAMERA CALIBRATION

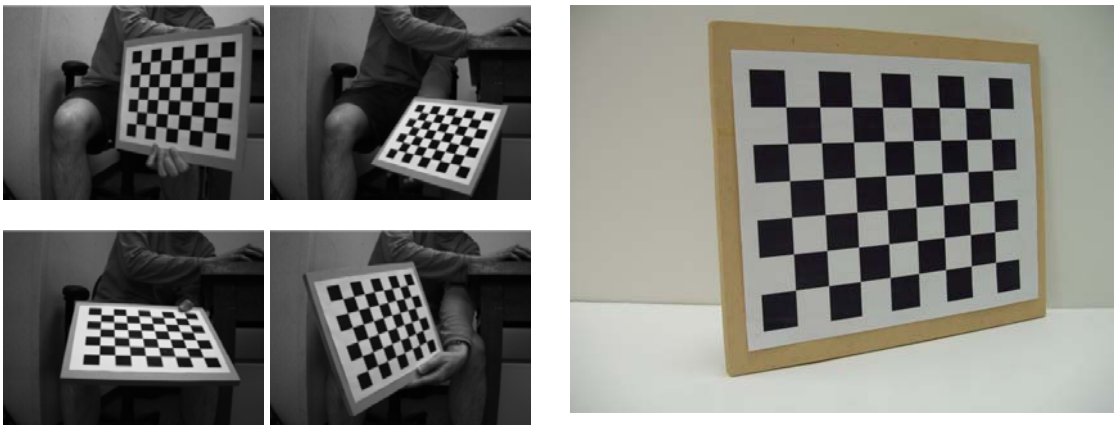


Figure 2. Sample calibration images (left) and the calibration target (right)

A camera calibration was performed using the *smallvcal* utility. Calibration is an automated process whereby a variety of intrinsic and extrinsic camera parameters are determined with a

high degree of accuracy. This is accomplished by taking pictures of a standard calibration target with known dimensions from different viewpoints (Figure 2), and then numerically finding a least squares solution.

The calibration parameters computed are shown in Table 2 and Table 3. The latter omits lens distortion parameters. A complete calibration file is provided for reference in as an appendix in 10.1.

Parameter	Description	Value (mm)
T_x, T_y, T_z	Translation from left to right imager	-88.72, -0.3257, 0.4584
R_x, R_y, R_z	Rotation from left to right imager	-0.0051, -0.0060, 0.0044

Table 2. Extrinsic camera calibration parameters

Parameter	Description	Value Left	Value Right
W, h	Imager width, height	640.0, 480.0 px	640.0, 480.0 px
C_x, C_y	Image center	275.9, 242.5 px	326.0, 250.9 px
f, f_y	Focal length	6.848, 6.850 mm	6.843, 6.853 mm

Table 3. Selected intrinsic camera calibration parameters for left and right imagers

After performing the calibration, the first step is to rectify the video images obtained for lens distortion. Rectification guarantees that straight lines will not appear curved in the camera image, and also that a given real-world object will appear on the same row of the left and right image (a property which we will use later during the feature matching step). The rectification is performed by a call to the *ReadParams("calibration.ini")* and *GetRect()* methods of the *svsVideoImages* object.

VII. ALGORITHM IMPLEMENTATION

7.1 OVERVIEW

The visual odometry algorithm can be summarized as the sequence of steps shown in Figure 3:

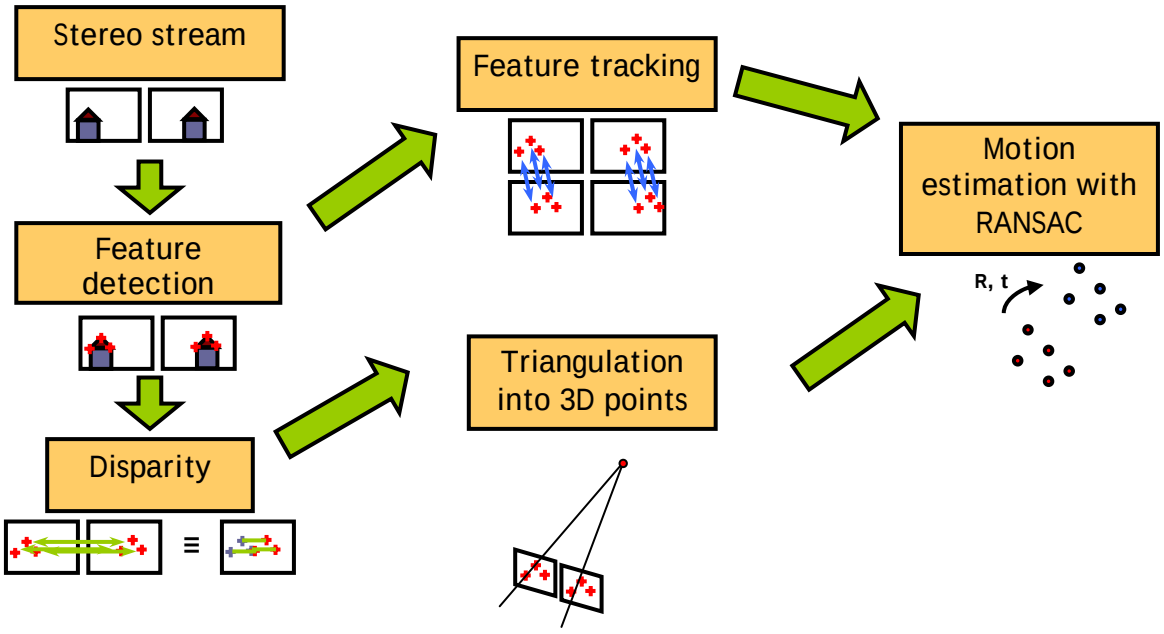


Figure 3. Visual odometry algorithm overview

The first step is to calculate corner features (good points to track in the video stream). Based on this, we can calculate disparity information (the difference in location of the same feature between the left and right image) for each feature and use it to triangulate features into 3D points in camera coordinates. Also, if a feature is tracked from frame to frame in a so-called feature track, we can use that information to see which 3D points match up from frame to frame. Finally, based on the movement of point clouds between frames, we can estimate the movement of the camera. Each step will now be explored in more detail.

7.2 DETECTING FEATURES

The first step in the system is to identify corner features in the left and right images. This is done using a method developed by Harris and Stephens, 1988[9]. This is done in the *harrisDetector* function, which takes an image and returns the (x, y) locations of up to a given number of features found in it. The function supports a fixed offset, which is excluded from the feature detection procedure on all four sides of the image. This is done in order to avoid artifacts due to the rectification procedure, and due to the camera imager. The implementation of this function was done largely by Prof. Bruce Maxwell as part of the Swarthmore Vision Module (SVM), and is reused here.

The feature detection process has five main steps, starting from the image in Figure 4:

1. Calculate derivatives and cross terms (Figure 5)
2. Calculate horizontal and vertical binomial filter on each derivative map (Figure 6)
3. Calculate the strength of corners (Figure 7)
4. Do non-maxima suppression to identify features (Figure 8)

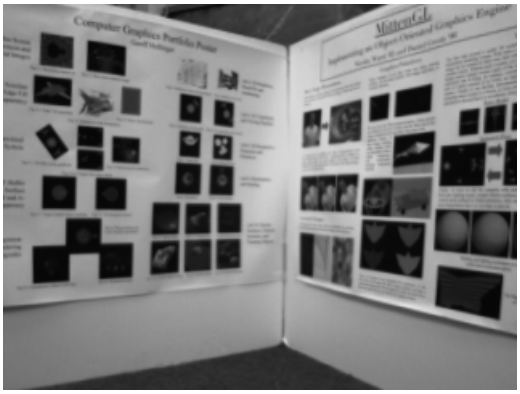


Figure 4. Original image

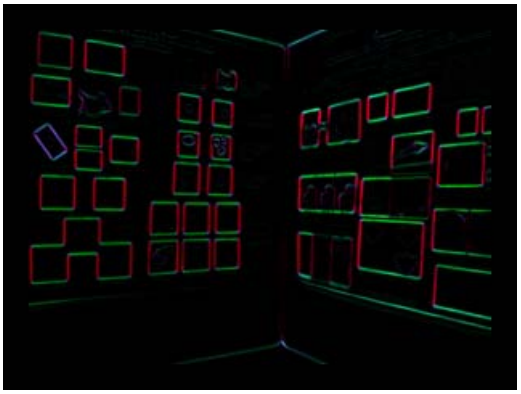


Figure 5. Image after derivatives and cross terms are calculated

Figure 6. Image after binomial filters

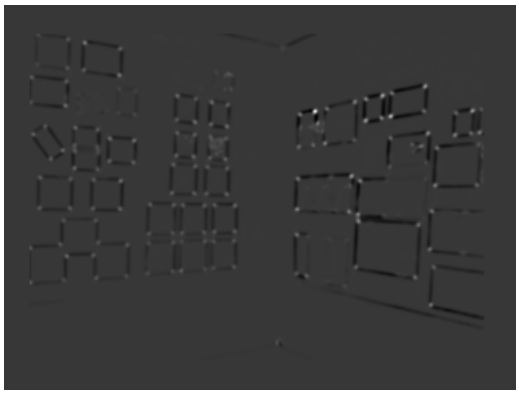


Figure 7. Image after determinant, trace, and strength calculation

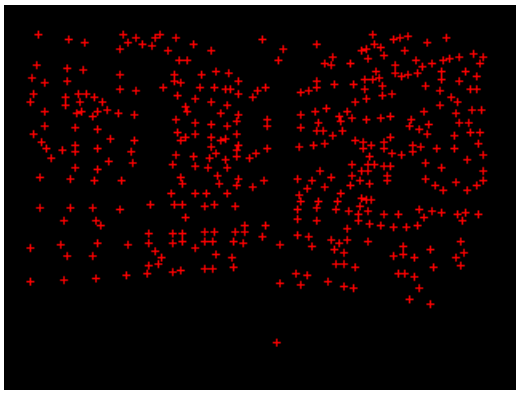


Figure 8. Image after non-maximum suppression

For a more robust implementation, OpenCV's *goodFeaturesToTrack* function was used to detect Harris corners, and the results were compared to this implementation. In addition to running a standard Harris detector, the OpenCV function removes all features that are within some threshold of strong features in an attempt to find robust features to track from frame to frame. Figure 9 demonstrates similar performance of both feature detectors, with the OpenCV detector perhaps exhibiting better accuracy (features are closer to corners in the image).

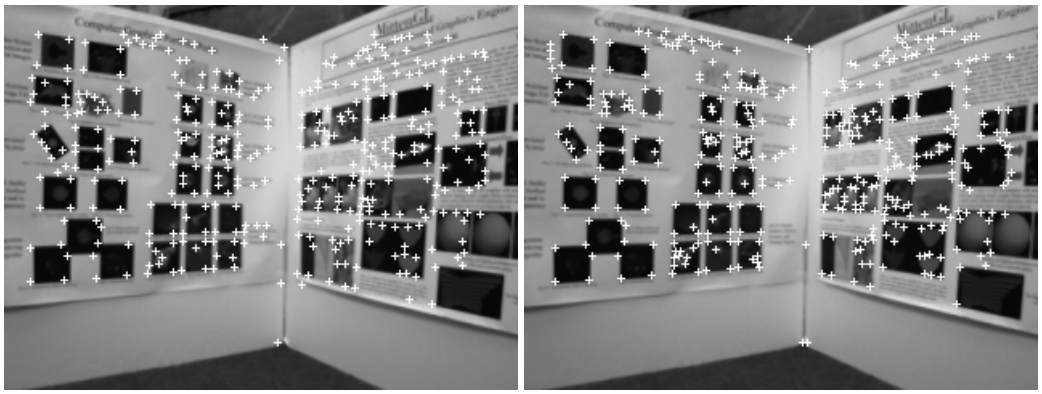


Figure 9. Comparing our Harris detector (left) with OpenCV's (right)

7.3 FEATURE MATCHING

After the Harris features in the image have been identified, we need to establish a mechanism to match a feature to its image in another image, regardless of whether we are working with a left-right image pair, or two consecutive frames from the left or right imagers.

The first step in feature matching is to establish a search square of a given size around the location where the given feature from the left image would map on to the right image. All the features within that square are considered to be potential matches. (Figure 10) The size of the square is empirically determined based on assumptions about the difference between the two views.

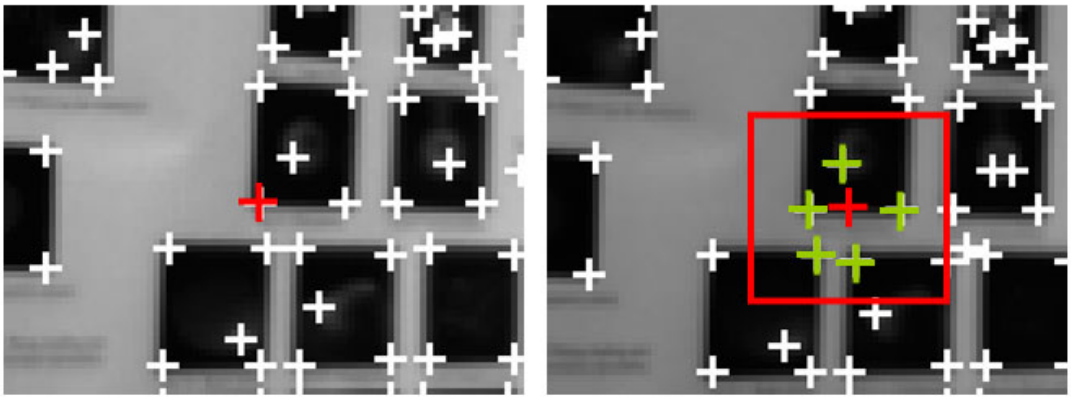


Figure 10. Feature matching

The next step is to compare the 11x11px patches surrounding the original feature on the left and all of its potential matches on the right. A normalized correlation is computed between all of them and the pair with the largest correlation is chosen to be the matching pair, as shown in Figure 11.

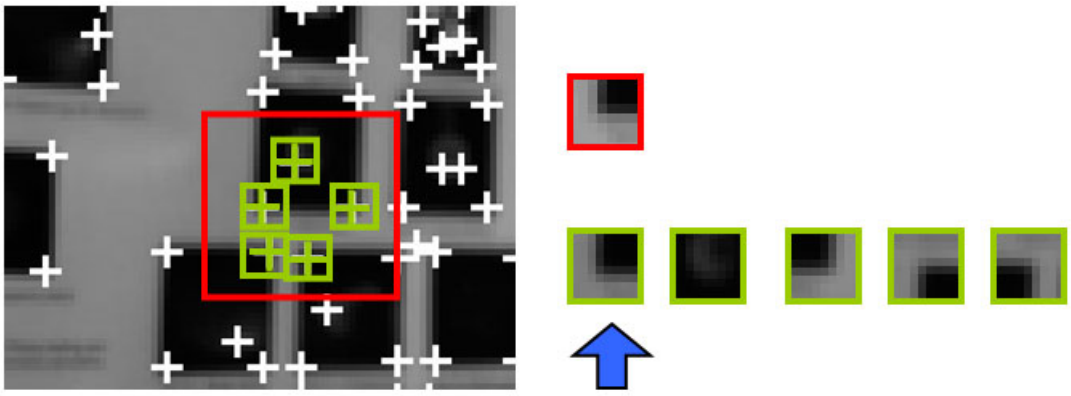


Figure 11. Calculating normalized correlation

The quantities A, B, and C are pre-computed for each patch according to the following equations:

$$\begin{aligned} A &= \sum I \\ B &= \sum I^2 \\ C &= \frac{1}{\sqrt{121B - A^2}} \end{aligned} \qquad \textbf{(1)}$$

where I corresponds to the intensity of a given pixel. At runtime, when comparing two patches, it is sufficient to calculate the quantity D

$$D = \sum I_1 I_2 \qquad \textbf{(2)}$$

and then the normalized correlation *n* between the two pixels is obtained by

$$n = (121D - A_1 A_2) C_1 C_2 . \qquad \textbf{(3)}$$

We perform this process from the left to right image and also from right to left, and only keep the bi-directional matches.

Feature matching is implemented in the *calculateDisparity* function.

7.4 FEATURE TRACKING

After a feature is detected, it is assigned to a so-called *feature track*. A feature track is a set of records of the location of a particular feature from frame to frame, up to some maximum number of frames. A track can receive up to a given number of *strikes*, which is the number of consecutive frames the feature may not appear before the entire track is deactivated. In this implementation, the feature track was 10 frames long, and it had up to 3 strikes before deactivation.

This is realized using the bi-directional feature matching process (7.3) used between consecutive frames on the left and on the right, with a search window of size 15x15px. The effect of this algorithm is illustrated visually in Figure 12.

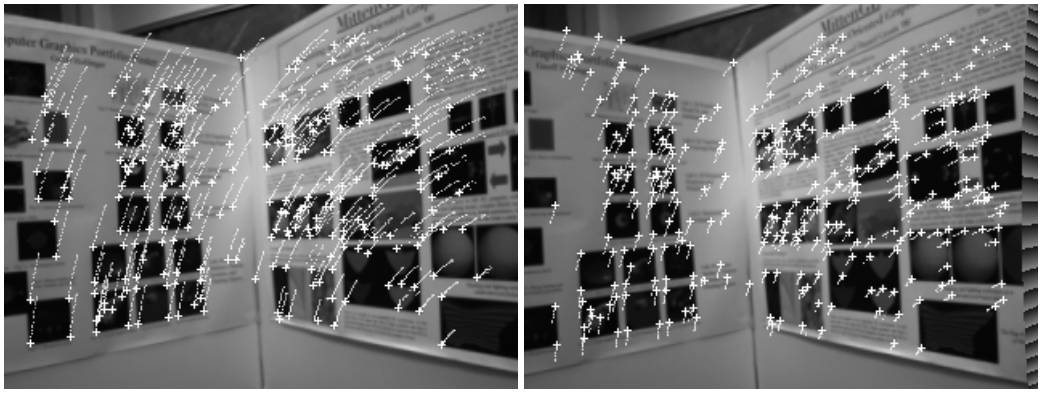


Figure 12. Feature matching frame-to-frame

This procedure is implemented in the *updateTracks* function, which in turn calls *calculateDisparity*.

In effect this process performs as follows. Every 10 frames, the feature detector generates up to a given number of tracks. As time passes, a large number of these tracks are deactivated, since the original feature that caused them to be created has disappeared or has not been visible for more than 3 consecutive frames. The number of tracks keeps dropping until it reaches some stable level, where most tracks are due to genuine strong features that can be traced for a prolonged period of time. At every frame, the feature detector will generate a number of features in excess of the number of active tracks, and most of those will not be associated with tracks and will subsequently be dropped. So we see that the feature tracking mechanism acts as a de-facto filtration system that eliminates spurious features due to noise. Also, it limits the speed at which the camera can move, since new feature tracks are generated only every 1/3 seconds, assuming a frame rate of 30 fps.

7.5 DISPARITY CALCULATION

Another necessary step, the importance of which will become obvious in the next section, is determining the matching features between the left and right images. Ideally, if we see a particular corner in both camera images, we want to know which feature on the left and which feature on the right correspond that corner.

We use thes bi-directional feature matching approach described above (7.3), only that this time we are comparing the left to the right image, instead of successive instances of the left or right image. Here a useful property of the rectification process (explained earlier) comes into play: a given point in the real world is guaranteed to be on the same row of the left and right images, with the only difference being in its horizontal position. This allows us to reduce the square search window to (in theory) a 40px-long linear search. However, to account for sub-pixel errors we add a pixel on each side and search in a 40x3px box. This concept is illustrated in Figure 13, where the right image shows the disparity map. Lighter points correspond to a larger disparity, which means the point is closer to the camera. The horizontal distance d ($0 \leq d \leq 40.11$), between the two features which correspond to the same world point, is called the *disparity* of that point. The significance of this distance is discussed below.

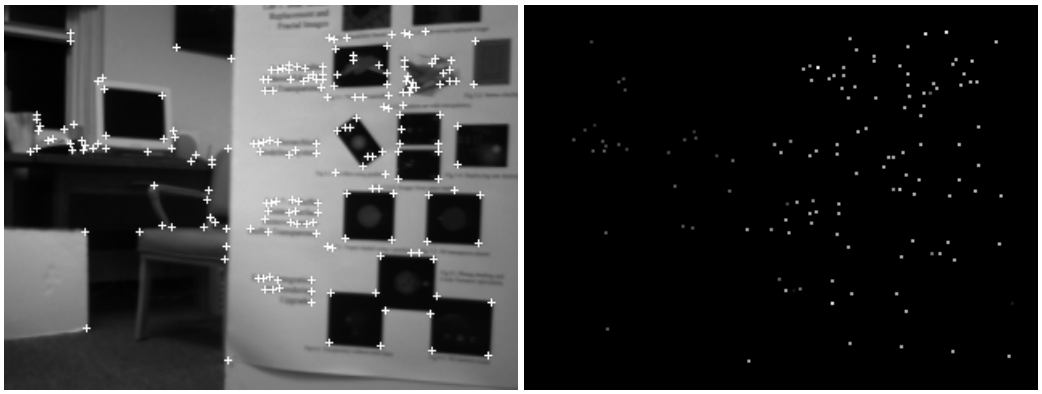


Figure 13. Feature matching left-to-right, with left image (left) and disparity map (right)

7.6 TRIANGULATING 3D POINTS

This step is the heart of the algorithm. Given the camera optical model and the parameters determined during the calibration step, we can turn a disparity into a distance in world coordinates.

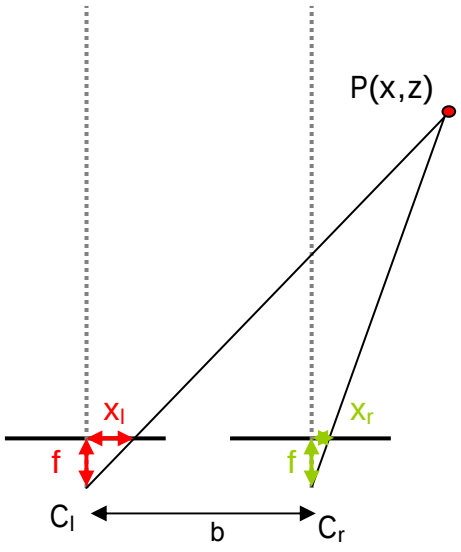


Figure 14. Triangulating disparity into 3D distance

Using similar triangles, and our knowledge of the intrinsic and extrinsic camera parameters (the quantities *f*, *b*, *Cl*, and *Cr* from Figure 14), we can calculate the quantities x and z using the following relationships:

$$z = \frac{bf}{x_l - x_r} = \frac{bf}{d}$$
$$\frac{x}{z} = \frac{x_l}{f} \quad \textbf{(4)}$$

Applying this idea in the xz and yz planes, we can triangulate the disparity information into a 3D point. An implementation of this geometric procedure exists as part of SVS so it was reused in order to save time. A *svsStereoProcess* object was created (an object that performs a variety of processing on a set of images), and *Calc3D()* method was invoked to convert disparity information to 3D points.

An example triangulation is shown in Figure 15.

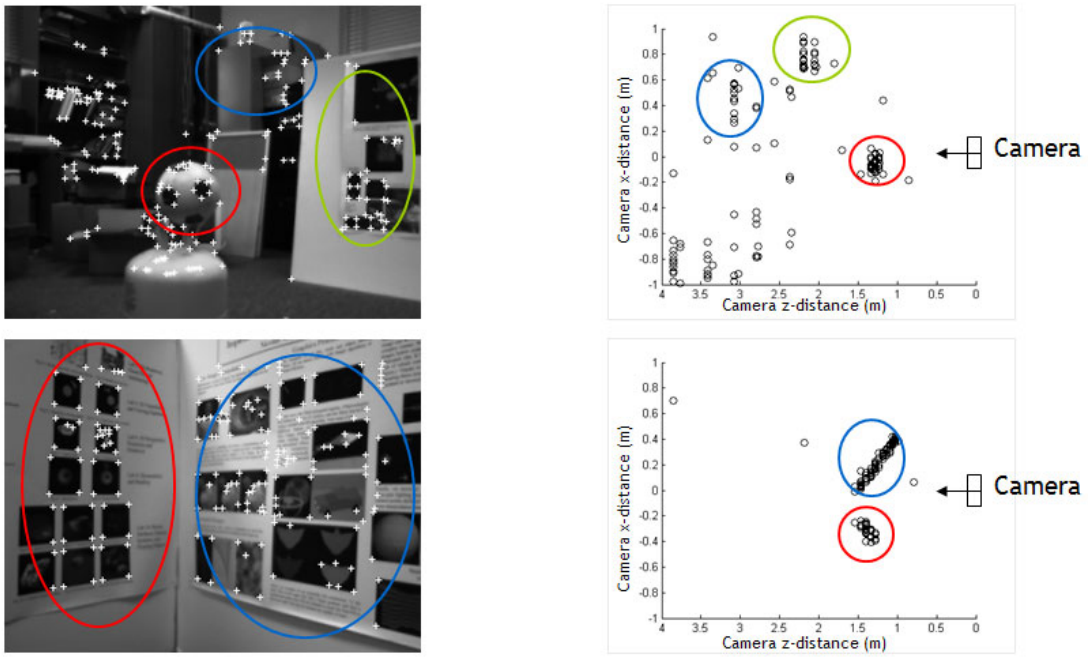


Figure 15. Sample triangulation results from a sparse (top) and a dense (bottom) environment

7.7 DETERMINING POINT CORRESPONDENCE IN TIME

At this point, we are able to triangulate disparity information to 3D points at every frame. For the pose estimation step, which will follow, we will need to be able to identify the 3D point corresponding to the same feature moving from the past to the current frame. In other words, we need to be able associate 3D points from frame to frame to recreate the movement of a particular 3D point from the point of view of the camera.

Solving this problem is the reason why we developed the feature tracking mechanism from frame to frame. Our feature tracks tell us how a feature moves in time, in the right or the left image. Identifying 3D point correspondences in time comes down to simply making sure that the points in the past and current frame originate from the same tracks. If two 3D points *p* and *p'* have been triangulated from features (f_L, f_R) and (f_L', f_R') (two in the past frame and two in the current one),

we can conclude that they show the movement of the same corner in the world in time if and only if f_L and $f_{L'}$ belong to the same track in the left image and f_R and $f_{R'}$ also belong to the same track in the right image.

This was implemented in the code as part of the *updateLocation* function.

7.8 POSE ESTIMATION

7.8.01 Summary

So far the implementation is able to triangulate features into 3D points at every frame and also determine the corresponding point in the previous frame for each point in the current frame. If there is any movement in the camera views, naturally these two point sets will be set off slightly from one another. We can also think about this situation as follows: in the past frame we had a 3D point cloud, which undergoes some movement, and is now visible in a new position in the current frame. So far we have been working in the reference frame of the camera, so we assume the features in the world are moving around the camera. This might be the case (if there is any movement in the scene as seen by the camera, e.g. a person walks by), but for the purposes of visual odometry task, we assume that most of the features seen by the camera are fixed with respect to the world reference frame. This is a reasonable assumption, as long as the majority of the features in the frame belong to stationary objects. Also, we can accommodate for some movement in the world, as we will see later. Assuming the 3D points are stationary in the world reference frame, any apparent movement of the 3D point clouds between frames, as perceived in the camera reference frame, is actually due to the movement of the camera itself. At this point we make the plausibility argument that by keeping track of the movement of a static point cloud from frame to frame, we can deduce the movement of the camera. We will develop this rigorously after developing some of the necessary pieces.

7.8.02 The Quaternion Motion Estimation Method

To calculate the movement of a point cloud, we need a minimum of three points and their positions before and after the move. We can think of the movement problem as the task of determining the rotation and translation that takes one triangle from one position to another. An important note: the two triangles will ideally be identical (no skewing or scaling), since each point corresponds to a feature in the world, and we are not permitting movement, let along shape distortion of the surrounding environment. We now formalize the idea of movement as follows

$$\vec{x}_i = R\vec{p}_i + t \tag{5}$$

where p_i is any point from the point cloud in the past frame, x_i is its corresponding point from the point cloud in the current frame, R is a rotation matrix, and t is a translation vector. To obtain the new location x_i of point p_i, we rotate it about some point (to be discussed later) by R and then translate by t. We can use a method developed by Besl[10] to solve for R and t.

Assume P and X are two triangles. First we calculate the "center of mass" of both triangles:

$$\vec{\mu}_p = \frac{1}{3}\sum_{i=1}^{3}\vec{p}_i \text{ and } \vec{\mu}_x = \frac{1}{3}\sum_{i=1}^{3}\vec{x}_i \tag{6}$$

The cross-covariance matrix Σ_{px} of the triangles P and X is:

$$\Sigma_{px} = \frac{1}{3}\sum_{i=1}^{3}[\vec{p}_i\vec{x}_i^{\,t}] - \vec{\mu}_p\vec{\mu}_x^{\,t} \tag{7}$$

We construct the quantities

$$A_{ij} = (\Sigma_{px} - \Sigma_{px}{}^{T})_{ij} \text{ and } \Delta = [A_{23} \quad A_{31} \quad A_{21}]^{T} \tag{8}$$

and we form the matrix

$$Q(\Sigma_{px}) = \begin{bmatrix} trace(\Sigma_{px}) & \Delta^{T} \\ \Delta & \Sigma_{px} + \Sigma_{px}{}^{T} - trace(\Sigma_{px}) I_3 \end{bmatrix} \tag{9}$$

The unit eigenvector $\vec{q}_R = [q_0 \quad q_1 \quad q_2 \quad q_3]^t$ corresponding to the largest eigenvalue of the matrix $Q(\Sigma_{px})$ can be used to yield *R* and *t*:

$$R = \begin{bmatrix} q_0{}^2 + q_1{}^2 - q_2{}^2 - q_3{}^2 & 2(q_1q_2 - q_0q_3) & 2(q_1q_3 + q_0q_2) \\ 2(q_1q_2 + q_0q_3) & q_0{}^2 + q_2{}^2 - q_1{}^2 - q_3{}^2 & 2(q_2q_3 - q_0q_1) \\ 2(q_1q_3 - q_0q_2) & 2(q_2q_3 + q_0q_1) & q_0{}^2 + q_3{}^2 - q_1{}^2 - q_2{}^2 \end{bmatrix} \tag{10}$$

$$\vec{t} = \vec{\mu}_x - R\vec{\mu}_p \tag{11}$$

A reference implementation of this algorithm was developed in MATLAB (Appendix 0) and eventually ported to C. A sample calculation is illustrated in Figure 16.

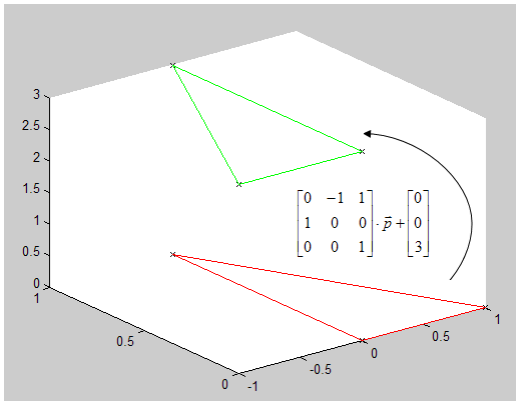


Figure 16. Sample movement estimation calculation

This functionality was implemented in the function *calculateRandT*.

Assuming an ideal point set, Besl's method would be the solution to the motion problem: pick any three points, and calculate R and t. However, our data is highly error-prone (especially along the dimension of the camera's optical ray), so just using this method would result in gross error that would render the implementation unusable. Therefore, we introduce RANSAC, a random-sample consensus algorithm, which is able to eliminate gross outliers and perform least-squares estimation on the valid data points.

7.8.03 The RANSAC Parameter Estimation Algorithm

RANAC was developed by Fischler and Bolles in 1981[11] as a robust alternative to least-squares fitting, which is immune to gross outliers (also known as poisoned data points). Applied to this particular problem, the RANSAC algorithm can be presented as the following sequence of steps:

1. Pick three points and use the 3-point problem solution presented above to calculate the *R* matrix and *t* vector.
2. Apply *R* and *t* to the entire past point cloud. If the transformation is perfect, the two sets should now overlap completely. That will not be the case, so we identify the points that are within a distance *e* of their positions in the current point cloud, and call them the *support set* for this particular hypothesis.
    a. If the support set has *t* or more members, then we call it the optimal set and move to step 3.
    b. If the support set has less than *t* members, we go back to step 1 and pick another hypothesis. We repeat this up to *k* times. If we cannot find a hypothesis with more than *t* members, then we pick the hypothesis with the largest support set to be optimal.
3. Once the optimal hypothesis is determined, we re-solve the model with the entire support set for that hypothesis. If we have more than 3 points in the support set, the system will be over-constrained, and we use a least-squares technique (described later) to come up with the polished *R* and *t*.

For this particular implementation, the following values were calculated for the RANSAC parameters, following the method outlined in Section II. A. and B. of the RANSAC paper:

Parameter	Definition	Value
e	Error tolerance in determining support	0.05m
t	Number of gross outliers	6
N	Number of points to select	3
W	Probability of an inlier	88%
SD(k)	Standard deviation on *k* value	0.83
K	Max. number of RANSAC runs	3

Table 4. RANSAC parameter values

The RANSAC algorithm was implemented as part of the *updateLocation* function.

7.8.04 SVD for Least Squares Solution of an Over-constrained System

In the last step of the RANSAC algorithm, we find a least-squares solution to an over-constrained system of equations. We set up the problem so we can use the singular-value decomposition (SVD) method, presuming the support set has *n* members:

$$A = \begin{bmatrix} & p_1^T & & 0 & 0 & 0 & 0 & 0 & 0 & \\ 0 & 0 & 0 & & p_1^T & & 0 & 0 & 0 & I_3 \\ 0 & 0 & 0 & 0 & 0 & 0 & & p_1^T & & \\ & & & & \cdots & & & & & \cdots \\ & p_n^T & & 0 & 0 & 0 & 0 & 0 & 0 & \\ 0 & 0 & 0 & & p_n^T & & 0 & 0 & 0 & I_3 \\ 0 & 0 & 0 & 0 & 0 & 0 & & p_n^T & & \end{bmatrix} \quad b = \begin{bmatrix} x_1 \\ \cdots \\ x_n \end{bmatrix} \quad x = \begin{bmatrix} R[0] \\ \cdots \\ R[12] \\ t \end{bmatrix} \qquad \textbf{(12)}$$

It is apparent that the solution to $Ax = b$ will yield the values of the elements of *R* and *t* for the full support set. Unfortunately, this equation cannot be solved because the system is over-constrained (it only takes 3 points to solve this uniquely), and the *A* matrix is not square and therefore not invertible.

To obtain an approximate (least-squares) solution, we run a numerical SVD on the *A* matrix, in order to solve the system for *x*:

$$[U\ \Sigma\ V] = SVD(A) \tag{13}$$

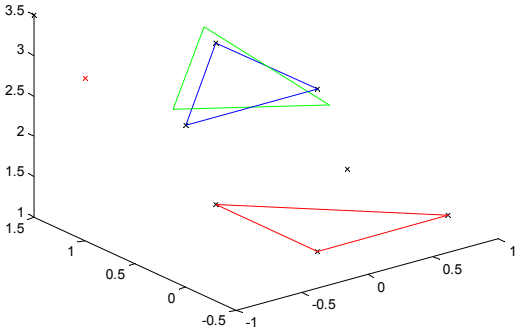
The diagonal matrix *Σ* contains the singular values for the matrix as its elements, while *U* and *V* contain a set of orthogonal output and input base-vector directions as their columns. We obtain the solution to the above system as follows:

$$x = V\Sigma^{+}U^{T}b \tag{14}$$

where Σ^{+} stands for the transpose of *Σ* with all nonzero entries replaces by their reciprocals.

This equation can be solved, and the elements of *x* make up *R* and *t* as shown in (12).

A reference implementation was done in MATLAB (Appendix 10.3), and some results are shown in Figure 17.



Original (red to blue)

$$R = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} t = \begin{bmatrix} 0 \\ 0 \\ 2 \end{bmatrix}$$

Over-constrained (red to green)

$$R = \begin{bmatrix} 0 & -1 & 0 \\ 1.25 & 0.25 & -0.0625 \\ 0.25 & 0.25 & 1.4375 \end{bmatrix} t = \begin{bmatrix} 0 \\ -0.0625 \\ 1.4375 \end{bmatrix}$$

Figure 17. SVD implementation

This was implemented in the *calculateSVD* function.

7.8.05 Software Implementation

The software was developed in C++, and compiled with gcc 3.2, with the optimization flags *–O3* and *–march=athlon*. The code is available in the CVS repository on *engin.swarthmore.edu.*

A pseudocode outline of the operation of the system follows:

```
(initialize camera and data structures)

(main loop) // runs until program is terminated

        updateTracks // update feature tracks with features in this frame
                calculateDisparity // used for feature matching between consecutive frames
```

```
        calculateDisparity // used to do feature matching between left and right frames

        Calc3D // triangulate into 3D points

        updateLocation // run RANSAC

                (figure out point correspondences)
                (RANSAC loop)
                        calculateRandT // used as RANSAC hypothesis generator
                (RANSAC loop end)
                calculateSVD // to solve best hypothesis with entire support set

(main loop end)
```

A detailed description of the API is provided as an appendix in 10.4.

VIII. TESTING

Figure 18. Test setup

The system was tested in lab conditions, using the setup shown in Figure 18.

The camera was positioned facing a set of posters, held upright on pieces of poster board. Masking tape was used to create a metric coordinate system on the floor, to facilitate measurement of the camera position.

The camera was mounted on a tripod, which had an adjustable height bar, which provided for easy measurement of vertical displacement (along the camera y-axis).

Instead of looking at a typical lab environment, the camera faces a controlled scene, since the system relies on good features for tracking and triangulation. Through testing we discovered that typical lab scene does not provide enough good features for tracking, and thus we opted for the posters, whose black-on-white corners were easy to detect and reliable in tracking. The shortcomings of the feature detector will be discussed further in the future work section.

8.1 VERIFYING BASIC FUNCTIONALITY

The first testing task was to make sure the system performs as expected. The system parameters were set as shown in Table 5.

The two most sensitive parameters in this implementation are the number of features used and the RANSAC matching distance, and we will later see how the former affects system performance.

Parameter	Value
Number of featrures	200
Disparity search width	40px
Disparity search height	2px
Tracking search width	15px
Tracking search height	15px
Track length	10 frames
Track strikes	3
Max camera FPS	30
RANSAC support matching distance	0.05m
RANSAC gross outliers	10
RANSAC maximum iterations	5

Table 5. System parameters used for testing

The first trial aimed at observing the steady-state error accumulated in the system. The camera was set facing the posters and the system was run, without causing any camera movement. No drift along any of the camera axes (a persistent position of (0, 0, 0)) would indicate good system performance.

No drift was observed along the camera x and y axes, but the position along the z axis drifted linearly with time as shown in Figure 19, with 16cm of drift being accumulated over 20 seconds, putting the drift velocity at 0.8 cm/s. This was clearly an unacceptable situation, so from this point on we only consider movement along the x and y axes. A further discussion of the drift along the optical axis is presented in the further work section.

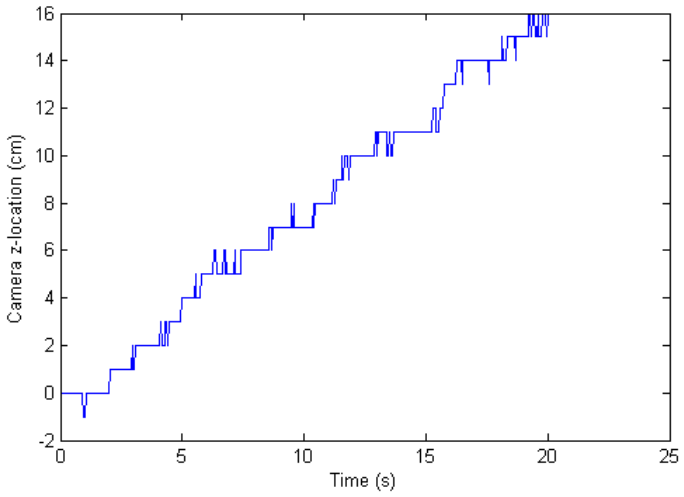


Figure 19. Camera drift along z-dimension

The second trial was conducted by moving the camera just along the x-axis (parallel to the ground): first 50 cm in the negative x direction in 10 seconds, followed by a 10-second pause, then 100 cm in the positive x direction in 10 seconds, then another 10-second pause, and finally back to the origin in 10 seconds. The results are shown in Figure 20.

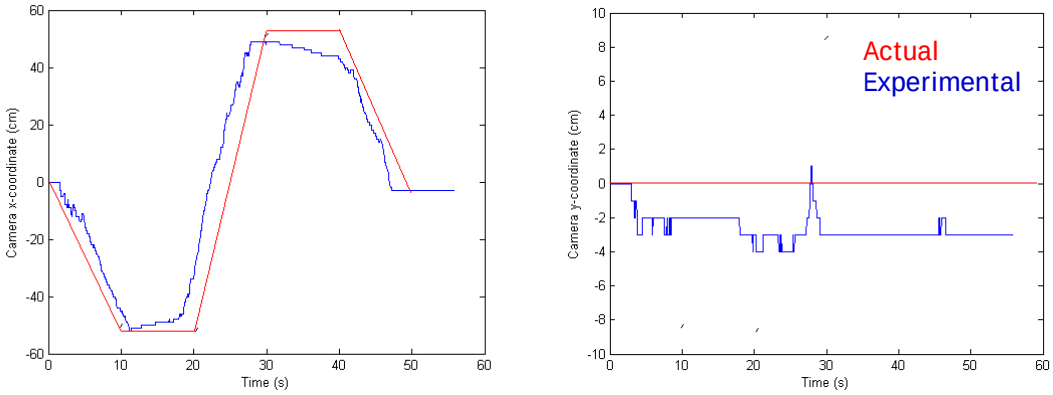


Figure 20. Camera movement along the x direction

We see the system performing adequately in the x direction, with a maximum error of 22cm. A persistent drift towards the origin is also noticeable if we look at the times when the camera is standing still. The y direction remains close to 0, with a maximum error of 4cm.

The same test was performed along the y (vertical) camera axis, except that the maximum displacement allowed by the tripod was 36cm, and the waiting time was reduced to about 5 seconds. The results are shown in Figure 21.

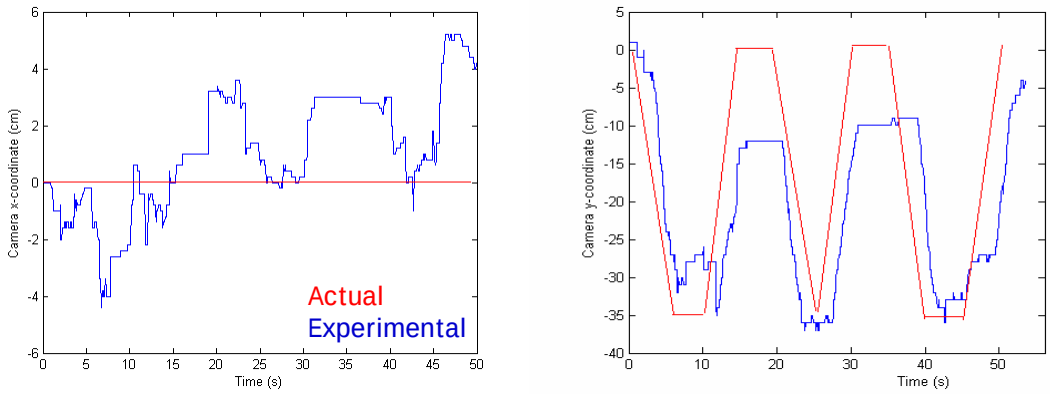


Figure 21. Camera movement along the y direction

Similar to the movement along the x axis, we see some noise in the dimension that's supposedly still: the maximum error in x is 5cm. The y-direction exhibits a maximum error of 12 cm, and again, we notice a persistent drift toward 0.

8.2 THE EFFECT OF FEATURE NUMBER

Varying the number of features the system attempts to detect in each image can drastically change its performance.

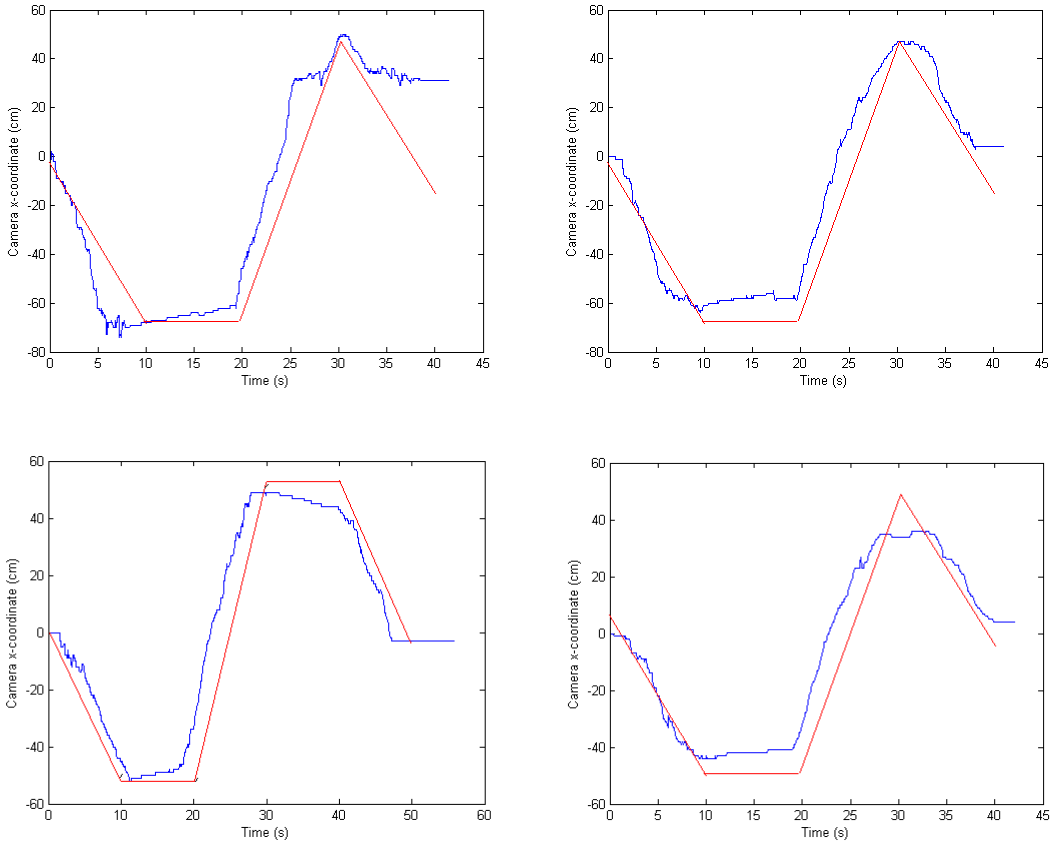


Figure 22. System performance with varying numbers of features and the resulting frame rate (clockwise 50/27.1, 100/27.3, 200/23.4, 500/10.5)

Figure 22 compares performance with a different number of features: 50, 100, 200, 500, 1000 (not shown). There exists an obvious trade-off between the number of features used and the frame rate: the more features the system has to process, the slower it goes. In trying to determine the optimal number of features, we notice that too low a number (50, 100) results in the system failing to stabilize at t=40s. Too many features, on the order of 1000, make the system too slow and there are not sufficient matches to run the RANSAC pose estimation. An important balance of abundant features, so that enough reliable movement information can be extracted about the environment, and a high enough frame rate, so enough many features can be processed at every frame: 200 – 600 features seems to be a good range.

The system performance satisfies our initial design constraints, with the exception of the drift in the z-dimension, which renders it unusable.

IX. FUTURE WORK

There are some clear shortcomings to the system:

- Slow processing speed, which prevents us from using large numbers of features
- A relatively slow camera movement speed
- Drift in the z-dimension, which makes it unusable

Some suggested improvements

- Clean up the code, and rewrite parallelizable components using the MMX instruction set. The system has lots of pixel processing, which is ideal for parallel processing. This would enable the system to run with more features, and also allow a faster camera movement speed.

- To counteract the drift in the z-dimension, a different method needs to be developed to estimate the camera pose. At present we track the movement of a triangle and use that to calculate an R and t matrices. However, the triangulated points have a high degree of uncertainty due to camera calibration errors and sub-pixel triangulation errors. Thus, that uncertainty is propagated to the motion hypothesis and causes drift, especially in the z-dimension, since triangulation produces the most uncertainty in the depth dimension. An alternative pose estimation method based on using a single camera and projections of image features (the 3-point problem)[12] is suggested in the original visual odometry paper[2]. This method limits the source of error to intrinsic camera parameters and if combined with reprojection error to determine the RANSAC support set, can result in more robust pose estimation (as explained in Section 4.2 of the paper). The OpenCV function *cvPOSIT*, which uses the POSIT algorithm to solve the 3-point-problem was integrated into the code, but there was not sufficient time to fully integrate the new method.

X. APPENDICES

10.1 SAMPLE CAMERA CALIBRATION

```
# SVS Engine v 4.0 Stereo Camera Parameter
File

[image]
have_rect 1

[stereo]
frame 1.0

[external]
Tx -88.760352
Ty -0.443364
Tz 0.960546
Rx -0.007640
Ry 0.004283
Rz -0.000043

[left camera]
pwidth 640
pheight 480
dpx 0.006000
dpy 0.006000
sx 1.000000
Cx 298.473769
Cy 256.148193
f 692.222883
fy 693.516532
alpha 0.000000
kappa1 -0.294415
kappa2 0.134338
kappa3 0.000000
tau1 0.000000
tau2 0.000000
proj
  6.940000e+02 0.000000e+00 3.020274e+02
0.000000e+00
  0.000000e+00 6.940000e+02 2.489592e+02
0.000000e+00
  0.000000e+00 0.000000e+00 1.000000e+00
0.000000e+00
rect
  9.999658e-01 5.103894e-03 -6.500436e-03
  -5.079060e-03 9.999797e-01 3.831156e-03
  6.519859e-03 -3.798009e-03 9.999715e-01

[right camera]
pwidth 640
pheight 480
dpx 0.006000
dpy 0.006000
sx 1.000000
Cx 333.638377
Cy 240.948377
f 692.706797
fy 694.477177
alpha 0.000000
kappa1 -0.311977
kappa2 0.186123
kappa3 0.000000
tau1 0.000000
tau2 0.000000
proj
  6.940000e+02 0.000000e+00 3.020274e+02 -
6.159968e+04
  0.000000e+00 6.940000e+02 2.489592e+02
0.000000e+00
  0.000000e+00 0.000000e+00 1.000000e+00
0.000000e+00
rect
  9.999290e-01 4.995068e-03 -1.082179e-02
  -5.036313e-03 9.999802e-01 -3.787431e-03
  1.080265e-02 3.841664e-03 9.999343e-01

[global]
GTx 0.000000
GTy 0.000000
GTz 0.000000
GRx 0.000000
GRy 0.000000
GRz 0.000000
```

10.2 MATLAB CODE FOR QUATERNION MOTION ESTIMATION

```
% This is supposed to solve analytically for the translation and rotation
% between two triangles in space

clear;
clc;

% syms p1x p1y p1z p2x p2y p2z p3x p3y p3z;
```

```
p1x = 0;
p1y = 0;
p1z = 0;
p2x = 2;
p2y = 0;
p2z = 0;
p3x = 0;
p3y = 1;
p3z = 0;

% syms x1x x1y x1z x2x x2y x2z x3x x3y x3z;
x1x = 0;
x1y = 0;
x1z = 0;
x2x = 0;
x2y = 2;
x2z = 0;
x3x = -1;
x3y = 0;
x3z = 0;

p1 = [p1x; p1y; p1z];
p2 = [p2x; p2y; p2z];
p3 = [p3x; p3y; p3z];

x1 = [x1x; x1y; x1z];
x2 = [x2x; x2y; x2z];
x3 = [x3x; x3y; x3z];

plot3(p1x, p1y, p1z, 'kx', p2x, p2y, p2z, 'kx', p3x, p3y, p3z, 'kx');
hold;
plot3(x1x, x1y, x1z, 'kx', x2x, x2y, x2z, 'kx', x3x, x3y, x3z, 'kx');
line([p1x; p2x; p3x; p1x], [p1y; p2y; p3y; p1y], [p1z; p2z; p3z; p1z], 'Color', [1 0 0]);
line([x1x; x2x; x3x; x1x], [x1y; x2y; x3y; x1y], [x1z; x2z; x3z; x1z]);

mup = [(p1x + p2x + p3x); (p1y + p2y + p3y); (p1z + p2z + p3z)]/3;
mux = [(x1x + x2x + x3x); (x1y + x2y + x3y); (x1z + x2z + x3z)]/3;

sigmapx = ((p1*x1.' - mup*mux.') + (p2*x2.' - mup*mux.') + (p3*x3.' - mup*mux.'))/3;

a = (sigmapx - sigmapx.');

delta = [a(2,3); a(3,1); a(1,2)];

q = [trace(sigmapx), delta.'; delta, sigmapx + sigmapx.' - trace(sigmapx)*eye(3)];

[eigvectors, eigvalues] = eig(q);

[maxvalue, index] = max(max(eigvalues));

qr = eigvectors(:,index);

R = [qr(1)^2+qr(2)^2-qr(3)^2-qr(4)^2, 2*(qr(2)*qr(3)-qr(1)*qr(4)),
2*(qr(2)*qr(4)+qr(1)*qr(3));
    2*(qr(2)*qr(3)+qr(1)*qr(4)), qr(1)^2+qr(3)^2-qr(2)^2-qr(4)^2, 2*(qr(3)*qr(4)-
qr(1)*qr(2));
    2*(qr(2)*qr(4)-qr(1)*qr(3)), 2*(qr(3)*qr(4)+qr(1)*qr(2)), qr(1)^2+qr(4)^2-qr(2)^2-
qr(3)^2]

t = mux - R*mup

p1 = R*p1 + t;
p2 = R*p2 + t;
p3 = R*p3 + t;

line([p1(1); p2(1); p3(1); p1(1)], [p1(2); p2(2); p3(2); p1(2)], [p1(3); p2(3); p3(3);
p1(3)], 'Color', [0 1 0]);
hold;
```

10.3 MATLAB CODE FOR SVD LEAST SQUARES SOLUTION

```
% This does SVD to find a least-squares solution to an overconstrained
% system.

clear;
clc;

% DECLARE ALL POINTS
% These were used to come up with R and t matrix
p1 = [0; 0; 1];
p2 = [1; 0; 1];
p3 = [0; 1; 1];

x1 = [0; 0; 3];
x2 = [0; 1; 3];
x3 = [-1; 0; 3];

% These are additional points, that were determined to be part of the
% support for this particular R and t
p4 = [1; 1; 1];

% R*p4 + t = [-1; 1; 3]. Perturb that a little (arbitrarily)
x4perfect = [-1; 1; 3];
x4 = [-1; 1.5; 3.5];

% Visualize some
plot3(p1(1), p1(2), p1(3), 'kx', p2(1), p2(2), p2(3), 'kx', p3(1), p3(2), p3(3), 'kx',
p4(1), p4(2), p4(3), 'kx');
hold;
plot3(x1(1), x1(2), x1(3), 'kx', x2(1), x2(2), x2(3), 'kx', x3(1), x3(2), x3(3), 'kx',
x4(1), x4(2), x4(3), 'kx');
plot3(x4perfect(1), x4perfect(2), x4perfect(3), 'rx');

line([p1(1); p2(1); p3(1); p1(1)], [p1(2); p2(2); p3(2); p1(2)], [p1(3); p2(3); p3(3);
p1(3)], 'Color', [1 0 0]);
line([x1(1); x2(1); x3(1); x1(1)], [x1(2); x2(2); x3(2); x1(2)], [x1(3); x2(3); x3(3);
x1(3)]);

% DEFINE R AND t - these were calculated between p1-3 and x1-3
% Rotation matrix
R = [0 -1 0; 1 0 0; 0 0 1]

% Translation vector
t = [0; 0; 2]

% DETERMINE REFINED R and t, CONSIDERING THE ADDITIONAL POINTS

% So we have A*x = b; where
% b is (4*3)x1, and contains all the points x
% x is 12x1, first 9 entries are the R elements, last 3 are the t elements
% A is (4*3)x12, and is derived by repeating p every 3 rows

b = [x1; x2; x3; x4];

% Helper matrices
p = [p1; p2; p3; p4];
I = eye(3);
f = zeros(3, 9);
A = [];

for i = 1:4,

    % Prepare left portion of matrix
    f = zeros(3, 9);
    f(1,1) = p(3*i-2);
    f(1,2) = p(3*i-1);
    f(1,3) = p(3*i);
    f(2,4) = p(3*i-2);
    f(2,5) = p(3*i-1);
    f(2,6) = p(3*i);
```

```
    f(3,7) = p(3*i-2);
    f(3,8) = p(3*i-1);
    f(3,9) = p(3*i);

    % Concatenate vertically until you get a (4*3) height
    A = [A; f, I];

end;

% Now do SVD on A
[U, S, V] = svd(A);

% Calculate S transpose...
S = S';

% ... and then go through and invert any non-zero elements
for i = 1:size(S,1),
    for j = 1:size(S,2),
        if S(i,j) < 1E-6,
            S(i,j) = 0;
        end;
        if S(i,j) ~= 0,
            S(i,j) = 1/S(i,j);
        end;
    end;
end;

% Finally, solve for new R and t
x = V*S*U'*b;
R = [x(1) x(2) x(3); x(4) x(5) x(6); x(7) x(8) x(9)]
t = [x(10); x(11); x(12)]

p1 = R*p1 + t;
p2 = R*p2 + t;
p3 = R*p3 + t;
p4 = R*p4 + t;

line([p1(1); p2(1); p3(1); p1(1)], [p1(2); p2(2); p3(2); p1(2)], [p1(3); p2(3); p3(3);
p1(3)], 'Color', [0 1 0]);
hold;
```

10.4 FULL API DESCRIPTION

```
/**
 * This function detects Harris corners
 *
 * @param[in] image The grayscale image to be analyzed for Harris corners
 * @param[in] height The image height
 * @param[in] width The image width
 * @param[in] maxFeatures The max number of features to be selected
 * @param[in] edgeBuffer The pixel offset to ignore from each image edge
 * @param[out] location The screen coordinates of the top features
 * @return The number of features detected, up to maxFeatures
 * @author Bruce Maxwell
 * @author Yavor Georgiev '06
**/

int harrisDetector(unsigned char *image, const int height, const int width, const int
maxFeatures, const int edgeBuffer, screenCoord location[]);

/**
* This function detects Harris corners and uses OpenCV.
*
```

```
* @param[in] image The grayscale image to be analyzed for Harris corners
* @param[in] height The image height
* @param[in] width The image width
* @param[in] maxFeatures The max number of features to be selected
* @param[in] edgeBuffer The pixel offset to ignore from each image edge
* @param[out] location The screen coordinates of the top features
* @return The number of features detected
* @author Yavor Georgiev
**/

int harrisDetectorCV(unsigned char *image, const int height, const int width, const int
maxFeatures, const int edgeBuffer, screenCoord location[]);

/**
 * This function calculates feature matches between two images, imageleft
 * and imageRight. You can use it to do either feature tracking or
 * disparity calculation.
 *
 * DISPARITY
 * The disparity and trackInfo outputs matter - leftMatches must be NULL
 *
 * FEATURE TRACKS
 * The leftMatches output matters - disparity and trackInfo must be NULL
 *
 * @param[in] imageLeft The left image
 * @param[in] imageRight The right image
 * @param[in] height The image height
 * @param[in] width The image width
 * @param[in] numFeaturesLeft The number of features detected in left image
 * @param[in] locationLeft The screen coordinates of the left features
 * @param[in] numFeaturesRight The number of features detected in right image
 * @param[in] locationRight The screen coordinates of the right features
 * @param[out] disparity The disparity image array
 * @param[out] trackInfo Track pairs for every disparity in the array
 * @param[out] leftMatches The coords of the corresponding feature on the right
 * @author Yavor Georgiev '06
 *
 **/

void calculateDisparity(unsigned char *imageLeft, unsigned char *imageRight, const long
height, const long width, const int window_height, const int window_width, const int
numFeaturesLeft, featureInfo locationLeft[], const int numFeaturesRight, featureInfo
locationRight[], short disparity[], trackPair trackInfo[], screenCoord leftMatches[]);

/**
 * This function updates the robot's location based on the points detected
 * in two consecutive frames
 *
 * @param[out] location The location vector to be updated
 * @param[in] currentPoints The points array for the current frame
 * @param[in] currentPointsTracks Tracks they came from
 * @param[in] currentPointsNum The number of members in the currentPoints array
 * @param[in] pastPoints The points array for the past frame
 * @param[in] pastPointsTracks The tracks they came from
```

```
 * @param[in] pastPointsNum The number of members in the pastPoints array
 * @param[in] R Array of rotation matrices for each hypothesis
 * @param[in] t Array of translation vectors for each hypothesis
 * @param[in] fsi The SVS stereo image this came from
 * @param[in] leftTrack Left feature tracks
 * @param[in] rightTrack Right feature tracks
 * @param[in] frame Current frame number
 * @author Yavor Georgiev '06
 *
 **/

void updateLocation(ColumnVector &location, svs3Dpoint currentPoints[], trackPair
currentPointsTracks[], int currentPointsNum, svs3Dpoint pastPoints[], trackPair
pastPointsTracks[], int pastPointsNum, Matrix **rotations, ColumnVector **translations,
svsStereoImage *fsi,featureTrack leftTrack[], featureTrack rightTrack[], short frame);

/**
 * This function updates the feature tracks given the features in the current
 * frame, and returns arrays that contain the feature coordinates from the
 * tracks in the current frame.
 *
 * @param[in] fsi Stereo image object, contains new video frame
 * @param[out] leftTrack The left feature track to update
 * @param[out] rightTrack The right feature track to update
 * @param[in] featuresLeft Number of features in the current left frame
 * @param[in] featuresRight Number of features in the current right frame
 * @param[in] bestFeaturesLeft Current left frame feature array
 * @param[in] bestFeaturesRight Current right frame feature array
 * @param[out] numFiltFeaturesLeft Corrected number of features on left
 * @param[out] numFiltFeaturesRight Corrected number of features on right
 * @param[out] filtFeaturesLeft Corrected left frame feature array
 * @param[out] filtFeaturesRight Corrected right frame feature array
 * @param[out] frame The current frame
 *
 * @author Yavor Georgiev '06
 *
 **/

void updateTracks(svsStereoImage *fsi, svsStereoImage *fsi_past, featureTrack
leftTrack[], featureTrack rightTrack[], int *numFiltFeaturesLeft, int
*numFiltFeaturesRight, featureInfo filtFeaturesLeft[], featureInfo filtFeaturesRight[],
short *frame);

/**
 * This function uses SVD to refine a given R and t hypothesis, considering
 * the entire support set for that particular hypothesis. There is a reference
 * MATLAB implementation of this somewhere.
 *
 * A haiku:
 * broken Camera
 * the World is beyond repair
 * it will not Work now
 *
 * @param[in] supportSize Size of support set
```

```
 * @param[in] currentPoints Support in current frame
 * @param[in] pastPoints Support in past frame
 * @param[out] R The polished rotation hypothesis
 * @param[out] t The polished translation hypothesis
 * @author Yavor Georgiev '06
 *
 **/

static void computeSVD(int supportSize, svs3Dpoint *currentPoints, svs3Dpoint
*pastPoints, Matrix &R, ColumnVector &t);

/**
 * This function uses the algorithm presented by Besl '92: "A Method for
 * Registration of 3-D Spapes" in section III.C., to find the best
 * rotation matrix and translation vector between two triangles in 3D space.
 *
 * @param[in] t1 An array of three points - the first triangle
 * @param[in] t1 An array of three points - the second triangle
 * @param[out] R The best rotation matrix between them
 * @param[out] t The best translation vector between them
 * @author Yavor Georgiev '06
 *
 **/

static void computeRandT(const Matrix t1, const Matrix t2, Matrix &R, ColumnVector &t);
```

XI. REFERENCES

[1] H. Choset, K. Nagatani, and N. Lazar, “The Arc-Transversal Median Algorithm: an Approach to Increasing Ultrasonic Sensor Accuracy,” in *Proc. IEEE Int. Conf. on Robotics and Automation*, Detroit, 1999.
[2] D. Nistér, O. Naroditsky, and J. Bergen, “Visual odometry,” in *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, June 2004.
[3] http://www.videredesign.com/docs/sthdcsg.pdf, May 06, 2006
[4] http://www.videredesign.com/small_vision_system.htm, May 06, 2006
[5] http://www.videredesign.com/docs/smallv-4.0.pdf, May 06, 2006
[6] http://www.videredesign.com/docs/calibrate-3.2.pdf, May 06, 2006
[7] http://www.robertnz.net/nm_intro.htm, May 06, 2006
[8] http://www.intel.com/technology/computing/opencv/, May 06, 2006
[9] C. Harris and M. Stephens, “A Combined Corner and Edge Detector,” *Proc. Fourth Alvey Vision Conf.,* pp. 147-151, 1998.
[10] Besl, P., McKay, N., "A Method for Registration of 3-D Shapes," *IEEE Trans. PAMI*, Vol. 14, No. 2, February, 1992, pp. 239-256
[11] Fischler, Martin A. and Robert C. Bolles, "Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography," *Communications of the ACM*, 24, 6 (1981), 381-395.
[12] Nister. A Minimal Solution to the Generalised 3-Point Pose Problem, *submitted to IEEE Conference on Computer Vision and Pattern Recognition*, 2004.