CS 322 Homework 3 - Solutions

out:	Thursday 22 February 2007
due:	**Wednesday 28 February 2007**

Part A - Half-Precision Arithmetic

Problem 1 What is the half-precision representation of the number 3? Of -27.5 in half-precision?

We can represent 3 in binary scientific notation as 1.1×2^1. The sign bit is 0, and the exponent is 1, which is biased by 15 to get 16. The mantissa is $1000.0000.00$ (note that there is an implicit 1). As a result, the half precision representation is $0.10000.1000000000$.

In binary scientific notation, -27.5 is -1.10111×2^4. The sign bit is 1, and the exponent is 4, which is biased by 15 to get 19. The mantissa is $1011.1000.00$ (again, note the implicit leading 1), and so the half precision representation is $1.10011.1011100000$.

Problem 2 What is the value of epsilon for $\frac{1}{10}$ in half-precision? For 1000.1 in half-precision?

Note: Epsilon is technically defined as the distance from the *floating point representation* of a given number to the next largest number in the same precision, not from the exact number to the next largest in the same precision. For instance, epsilon of $\frac{1}{10}$ is the distance from the floating point representation of $\frac{1}{10}$ (which will not be exactly $\frac{1}{10}$) to the next largest number in the same precision.

For $\frac{1}{10}$, we observe that it can be represented as $\frac{1}{16} + \frac{1}{32} + \frac{1}{256} + \frac{1}{512} + \frac{1}{4096} + \frac{1}{8192} + \ldots$, or $1.1001100110 \times 2^{-4}$ in half precision. The last bit in the mantissa represents 2^{-14}, so the next largest number is 2^{-14} away in half precision, and so 2^{-14} is the value of epsilon for $\frac{1}{10}$.

For 1000.1, we observe that it can be represented as 1.1111010000×2^9 in half precision. Again, the last bit in the mantissa represents 2^{-1}, so the next largest number is 2^{-1} away in half precision, and so 2^{-1} is the value of epsilon for 1000.1.

Part B - Error Visualization

Problem 1

Problem 2

```
function plotError(data)
% Takes in a matrix of data from roundoffPlotter and
% generates one or more interesting plots of the error.
```

```
% See the homework assignment for a detailed description
% of the format of data.

% Create a new figure to draw in.  You SHOULD NOT remove them.
figure();
axes();

% Your code to plot the error should go here

% create a subplot with 4 plotting regions, set the first one
% to be active
subplot(2, 2, 1);

numSteps = size(data, 2);
angles = 1:numSteps;
radians = angles*pi/180;

% compute the error in edge lengths for each edge
e1 = data(3:4, :) - data(1:2, :);
e2 = data(5:6, :) - data(3:4, :);
e3 = data(7:8, :) - data(5:6, :);
e4 = data(1:2, :) - data(7:8, :);

len1 = abs(sqrt(sum(e1 .* e1, 1)) - sqrt(2));
len2 = abs(sqrt(sum(e2 .* e2, 1)) - sqrt(2));
len3 = abs(sqrt(sum(e3 .* e3, 1)) - sqrt(2));
len4 = abs(sqrt(sum(e4 .* e4, 1)) - sqrt(2));

% plot using standard line graph
plot(angles, len1, 'r', angles, len2, 'g', ...
     angles, len3, 'b', angles, len4, 'm');
title('Magnitude of error in edge lengths over time');
legend('Edge1 (p2-p1)', 'Edge2 (p3-p2)', ...
        'Edge3 (p4-p3)', 'Edge4 (p1-p4)', ...
        'Location', 'NorthWest');
set(get(gca, 'XLabel'), 'String', 'Iteration number');
set(get(gca, 'YLabel'), 'String', 'Magnitude of error');

% set the second subplot region to be active
subplot(2, 2, 2);

% plot the same data, except in a polar plot
polar(radians, len1, 'r');
hold on
polar(radians, len2, 'g');
polar(radians, len3, 'b');
polar(radians, len4, 'm');
hold off
```

```
title('Magnitude of error in edge lengths as a function of angle');
legend('Edge1 (p2-p1)', 'Edge2 (p3-p2)', ...
       'Edge3 (p4-p3)', 'Edge4 (p1-p4)', ...
       'Location', 'SouthEast');
set(get(gca, 'XLabel'), 'String', 'Angle');
set(get(gca, 'YLabel'), 'String', 'Magnitude of error');

% set third subplot region to be active
subplot(2, 2, 3);
trueVals = zeros(8, numSteps);
trans = (data(5, 1) + data(1, 1))/2;

% compute the rotation using double exact for
% the given settings
for i = 1:numSteps
    amt = i*pi/180;
    rotMat = [cos(amt) -sin(amt) 0; sin(amt) cos(amt) 0; 0, 0, 1];

    affPts = [trans, trans-1, trans, trans+1; ...
              trans+1, trans, trans-1, trans; ...
              ones(1, 4)];

    rotMat = [1, 0, trans; 0, 1, trans; 0, 0, 1] ...
                * rotMat ...
                * [1, 0, -trans; 0, 1, -trans; 0, 0, 1];
    tmpPts = rotMat*affPts;
    trueVals(:, i) = reshape(tmpPts(1:2, :), 8, 1);
end

% Compute distance of each point from double exact
% location
trueD1 = data(1:2, :) - trueVals(1:2, :);
trueD2 = data(3:4, :) - trueVals(3:4, :);
trueD3 = data(5:6, :) - trueVals(5:6, :);
trueD4 = data(7:8, :) - trueVals(7:8, :);

errD1 = sqrt(sum(trueD1 .* trueD1, 1));
errD2 = sqrt(sum(trueD2 .* trueD2, 1));
errD3 = sqrt(sum(trueD3 .* trueD3, 1));
errD4 = sqrt(sum(trueD4 .* trueD4, 1));

plot(angles, errD1, 'r', angles, errD2, 'g', ...
     angles, errD3, 'b', angles, errD4, 'm');
title(sprintf(['Magnitude of error in positions over time\n' ...
               ['(compared to double exact)']));
legend('P1', 'P2', 'P3', 'P4', 'Location', 'NorthWest');
set(get(gca, 'XLabel'), 'String', 'Iteration number');
set(get(gca, 'YLabel'), 'String', 'Magnitude of error');
```

```
% Calculate dot product of each edge with subsequent
% edge (obtaining cosine of angle between them)
angle1 = abs(sum(e1 .* e2, 1));
angle2 = abs(sum(e2 .* e3, 1));
angle3 = abs(sum(e3 .* e4, 1));
angle4 = abs(sum(e4 .* e1, 1));
subplot(2, 2, 4);
plot(angles, angle1, 'r', angles, angle2, 'g', ...
     angles, angle3, 'b', angles, angle4, 'm');
title(sprintf(['Magnitude of error in cos(angle)\n'...
                'between adjacent edges']));
legend('Edges 1-2 (p3-p2-p1)', 'Edges 2-3 (p4-p3-p2)', ...
        'Edges 3-4 (p1-p4-p3)', 'Edges 4-1 (p2-p1-p4)', ...
        'Location', 'NorthWest');
set(get(gca, 'XLabel'), 'String', 'Iteration number');
set(get(gca, 'YLabel'), 'String', 'Magnitude of error');
```

Each figure consists of four plots of the error for a particular setting. The top left plot measures the magnitude of the error in each of the four edge lengths of the square; that is, how much each edge deviates from the expected value of $\sqrt{2}$. The top right plot looks at the same data, except organized as a polar plot by the total angle rotated. The bottom left plot looks at how far each point deviates from the exact double precision result, which can be considered to be the best possible answer obtainable with the current options. Finally, the bottom right plot looks at the absolute value of the cosine of the angle between each of neighboring edges; since angles should be preserved by a rotation, the edges should remain perpendicular and so the cosine of the angle (i.e. the dot product of the two vectors) should be 0.

The first thing we note is the general increasing trend in the cumulative mode for both single and double precision. For the time period analyzed, this growth appears to be roughly linear, with the double precision result being about 10^{-9} more accurate in the lengths and positions. We should also note that the bottom right plot for single precision appears to have an issue with the axes limits; however when we inspected the data it turned out that the cosine of the angles was in fact 0 at all points in time (i.e. no error). It is unknown precisely why this is the case; one possible explanation is that since the points are symmetrical, they are traversing the same sequence of available floating point numbers, which could also explain why the error measures in the lengths are the same for all four edges. However, since we don't see this behavior for double precision, it bears further investigating.

When we switch to exact mode at the origin, we can see that any upward trends in the error are lost. This is unsurprising, as exact mode does not compound the errors made by previous steps, and so we expect to see in general a relatively constant, small rate of error. There are some periodic effects present, possibly due to the repetition of the same angle over time.

When we move to single precision cumulative about $(10000, 10000)$, we see that the error is compounded greatly, and visual artifacts are very apparent. The lengths of edges change dramatically, and the perpendicularity of neighboring edges is lost completely. This only gets worse as the point is pushed further and further out, finally not moving at all around $(10000000, 10000000)$ (see Problem

3 for more details). When we move to exact mode, single precision, at $(1000000, 1000000)$, there are readily apparent visual artifacts, as the square jumps around noticeably on screen. However, because the error isn't cumulative, the square still spins, albeit in a very erratic fashion. Moving to double precision cumulative, we see that the expanded number of bits in the mantissa aids greatly in this range, as the overall error is less than single precision exact, and is below the range for visual detection.

Problem 3 At about 1e7 in single precision, the square stops rotating entirely. If we increase this to about 2e7 we see that the square seems to disappear entirely. This makes sense if we look at the value of epsilon in single precision at these distances from the origin; at 1e7, epsilon is 1. This means that single precision is only capable of distinguishing between integers at this point – there is no room in the mantissa for any fractional value. As a result, when we attempt to rotate the square slightly all of the fractional information is lost when we translate back to $(1\text{e}7, 1\text{e}7)$, leaving the square stuck at its original integer amounts (and thus not moving). At the larger value of 2e7, epsilon is now 2, and there are not enough bits in the mantissa to even distinguish between neighboring integers. As a result, the four corners of our square end up at the same floating point value, and the square has collapsed into a single point.

Problem 4 In general, we can expect that the error in our computation at each step will be roughly proportional to the value of epsilon at 1 in our chosen precision, since in some sense that is a measure of the amount of information we "lose" at each step. Looking at our plots, we can see that single precision cumulative at the origin has an error in lengths of about $4.2\text{e}-5$ after 2500 iterations, or about $352.3 \times \text{eps}(\text{single}(1))$. Similarly, for double precision cumulative at the origin, the error in lengths is about $1.1\text{e}-13$ after 2500 iterations, or about $495.4 \times \text{eps}(1)$. Given these, we can generate a ballpark guess of the error introduced by half precision as somewhere inbetween 300 to 500 times $\text{eps}(\text{half}(1))$, or about 0.29 to 0.49. In general, though, this is just a back of the envelope estimate, but it seems to indicate that we should expect to see errors around 0.1 to 1 in magnitude, which is a significant (and visually noticeable) amount of error.

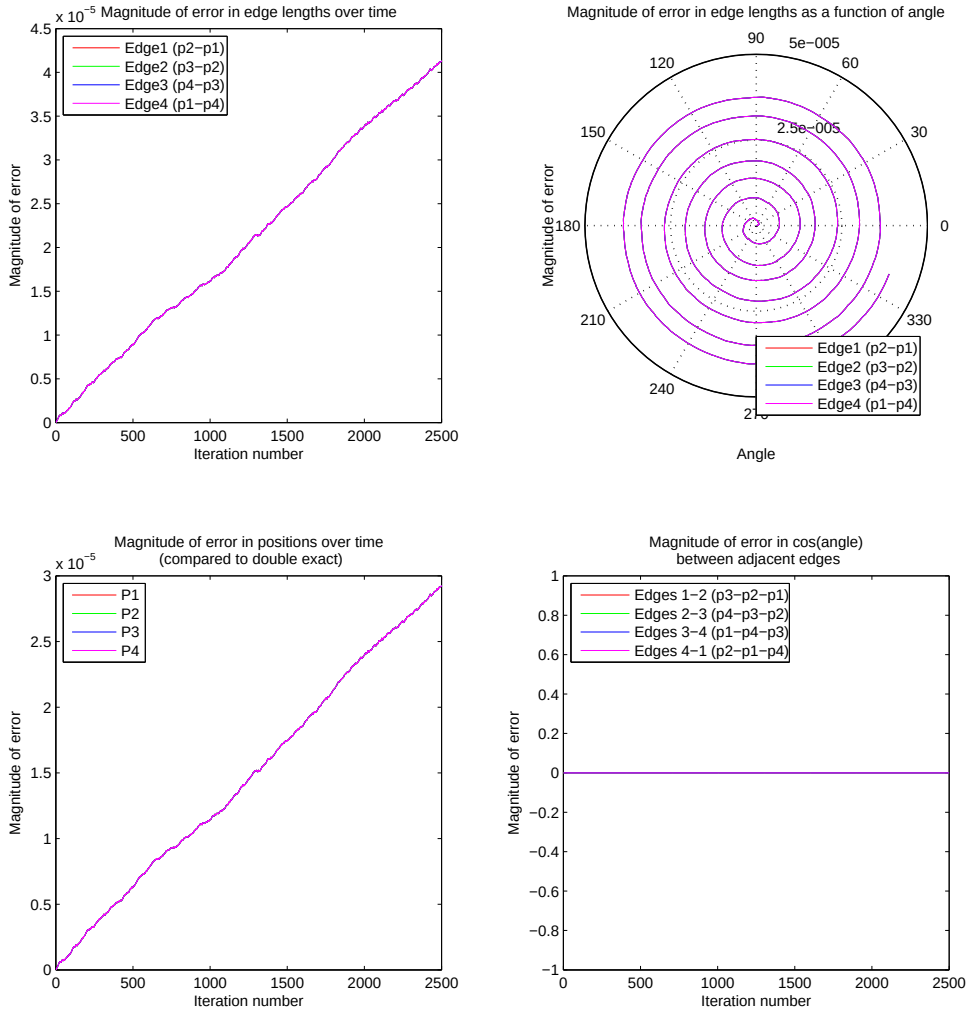

Figure 1: Graphs of the output of the program in single precision, cumulative rotations, about the origin

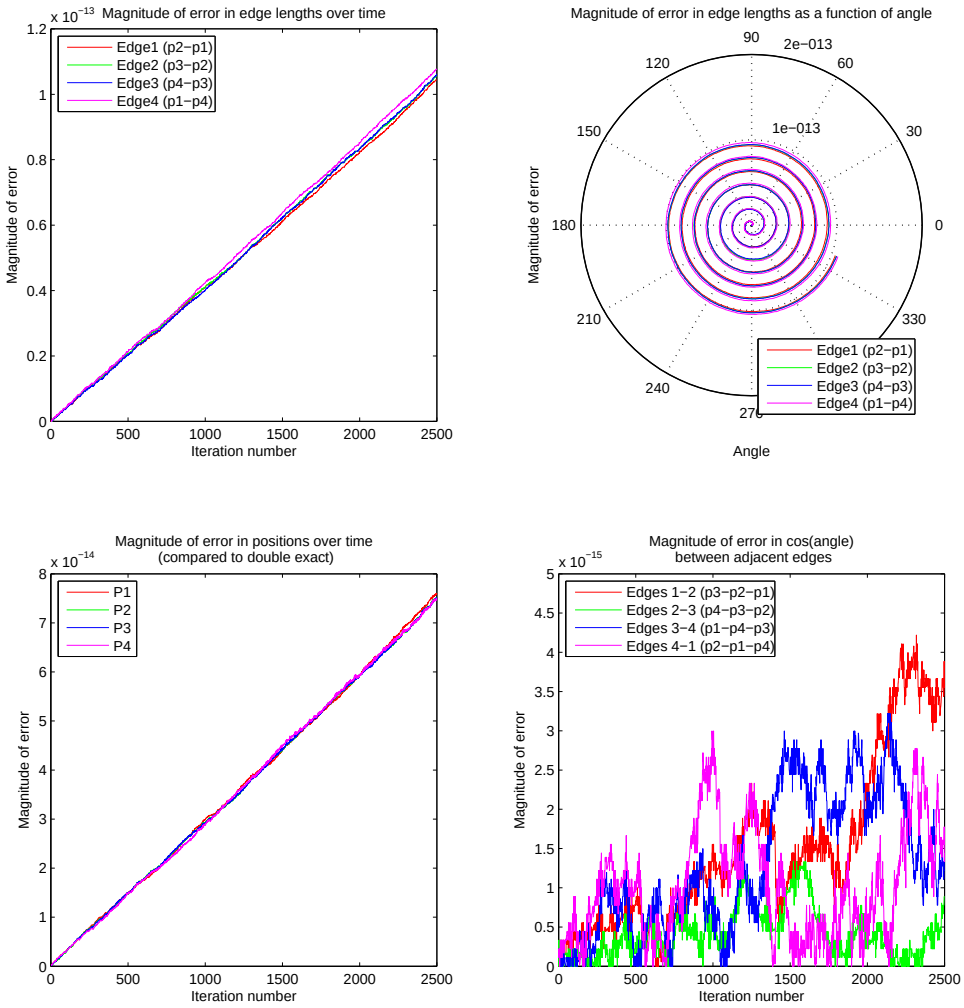

Figure 2: Graphs of the output of the program in double precision, cumulative rotations, about the origin

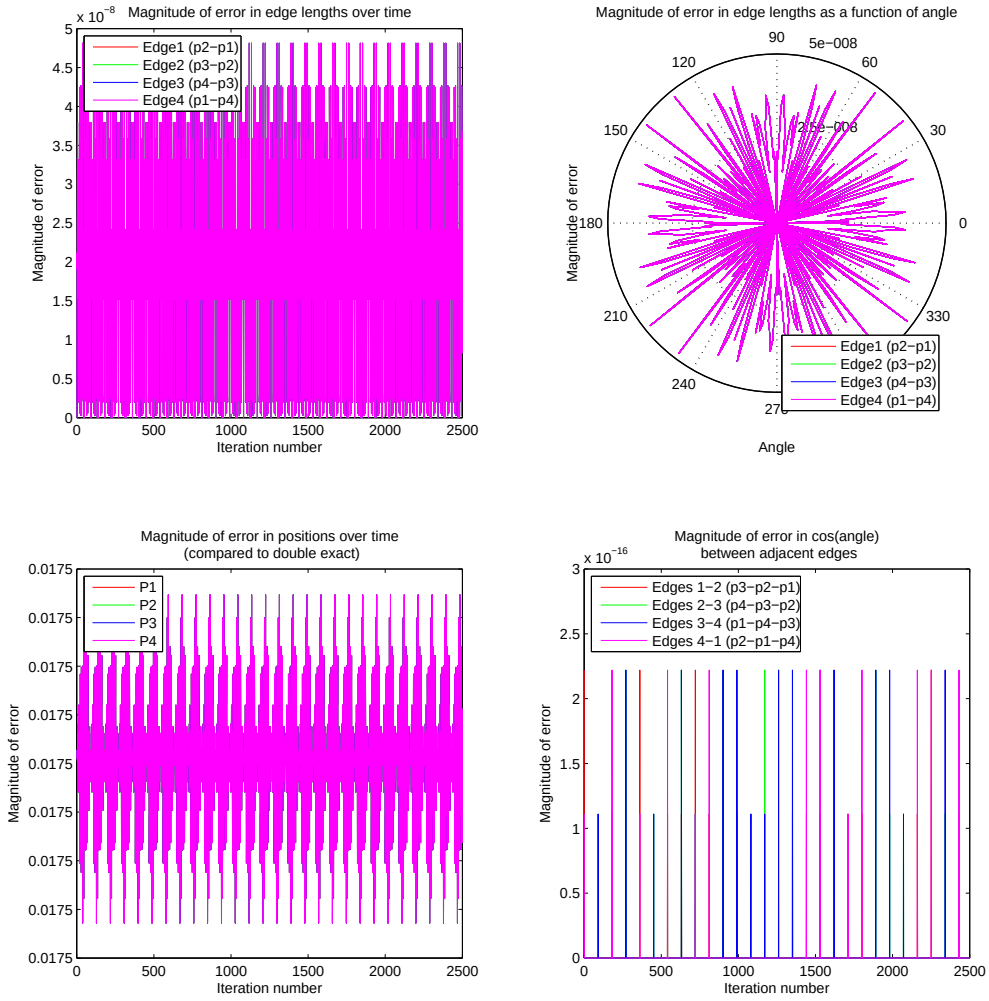

Figure 3: Graphs of the output of the program in single precision, exact rotations, about the origin

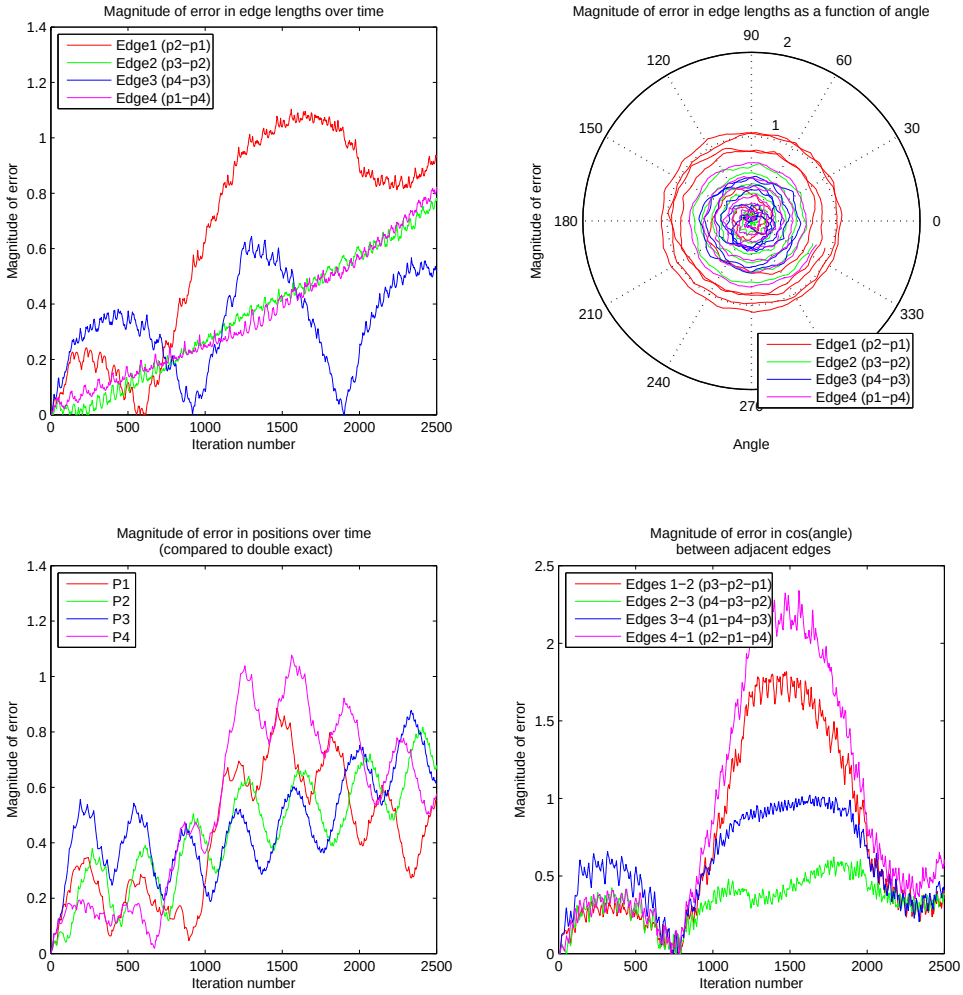

Figure 4: Graphs of the output of the program in single precision, cumulative rotations, about the point $(100000, 100000)$

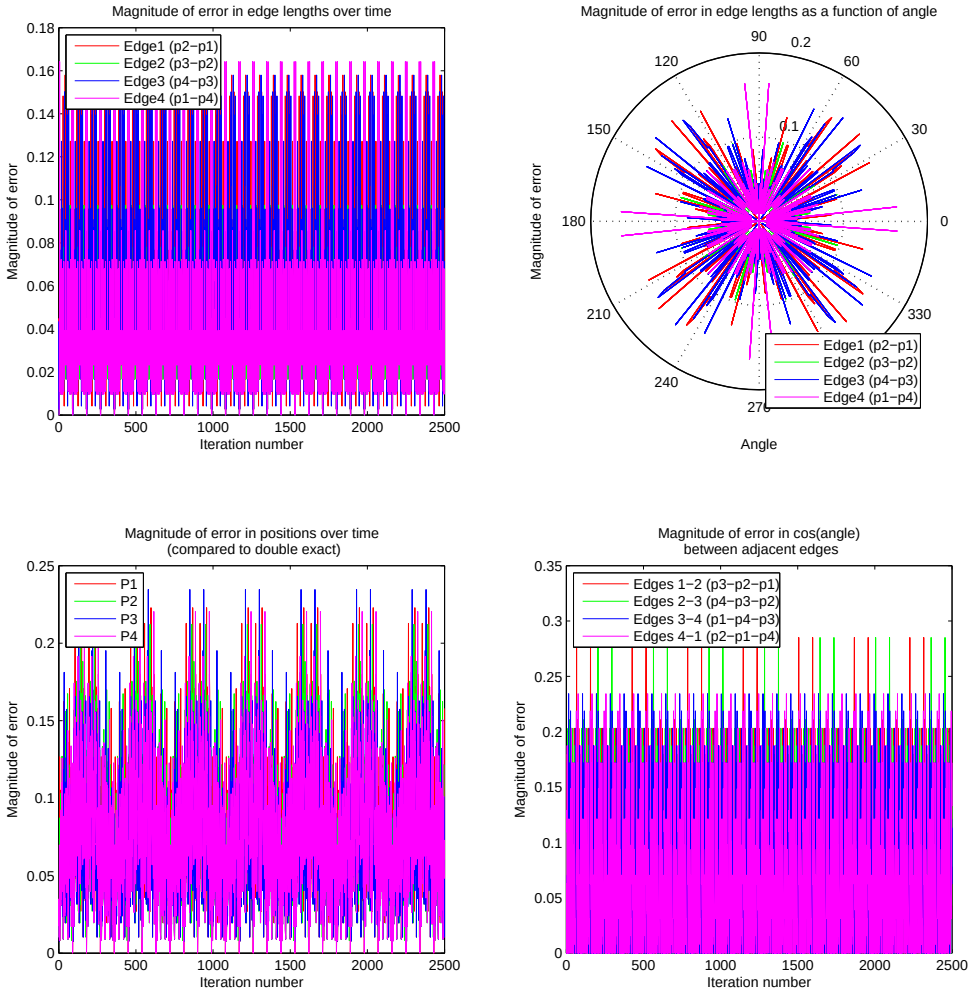

Figure 5: Graphs of the output of the program in single precision, exact rotations, about the point $(1000000, 1000000)$

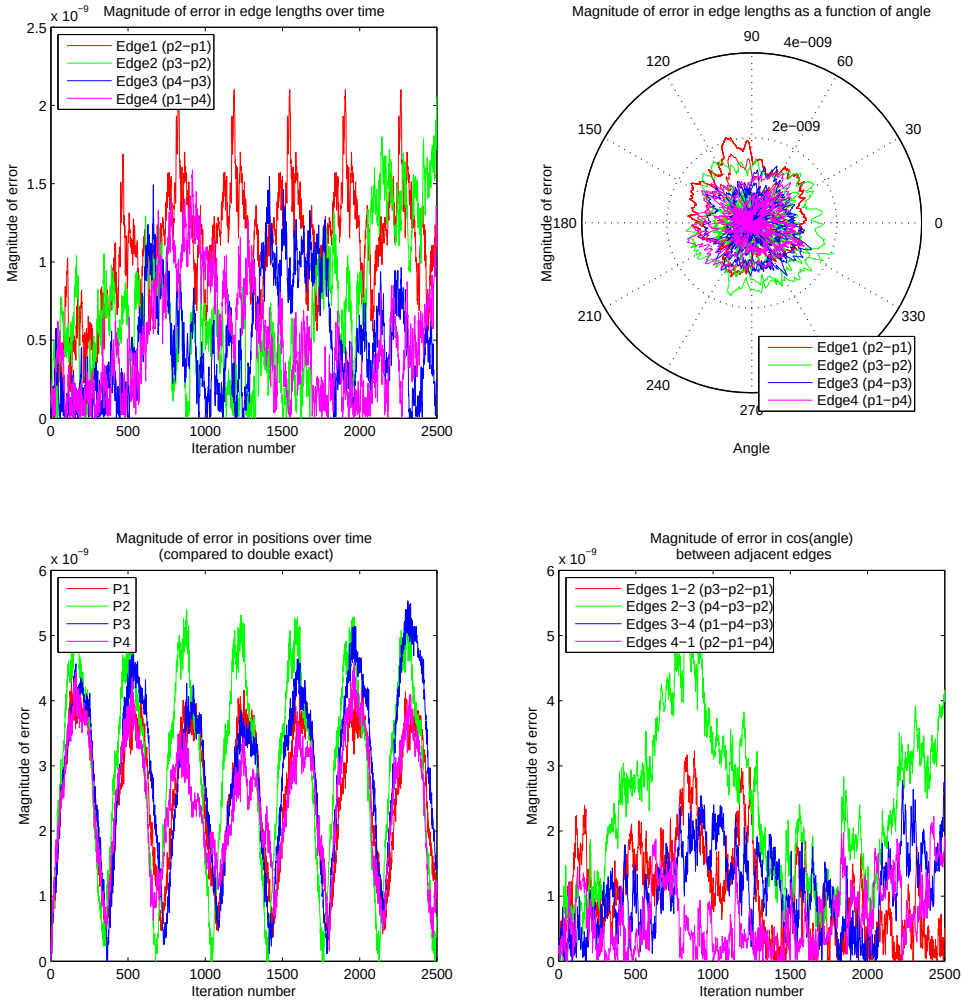

Figure 6: Graphs of the output of the program in double precision, cumulative rotations, about the point $(1000000, 1000000)$