A Relative Point Algorithm

Jay Kraut

Abstract— **This paper presents a Relative Point Algorithm. The Relative Point algorithm uses 3D points as input. Multiple observations of the same point are stored untransformed in a data structure. The previous iterations untransformed points are used to register the next iteration's points, neglecting the use of current position for registration. Points that are visible at the same time interval are grouped together. A relative map of each group is computed from the stored untransformed point's average distance from an axis computed from three basis points. The use of the basis points allow the average to be translation and rotation invariant. The relative maps are combined using points present in multiple groups with least squared fitting. Current position is generated at the end of each iteration by least square fitting the current observation of points to their globally map positions. Current position is used locally for backtracking and only globally for closing the loop. It is possible to not use current position at all and instead rely on point merging that has a higher computation overhead. The untransformed observations are also used in future computation to form better groupings and dynamic point detection. The Relative Point algorithm is shown to have $O(n_a \log n_a)$ computation complexity where n_a is the average quantity of points seen. Its average run time given an average of 100 observed points per iteration is two milliseconds. The total quantity of points does not affect the run time. The accuracy of the Relative Point algorithm is shown to be comparable to a 6D no odometry EKF in a simulation using white Gaussian noise, and is able to identify dynamic points in O(nlogn) time.**

I. Introduction

This paper works on the Simultaneous Localization and Mapping (SLAM) problem using simulated 3D points as input and no odometry. Algorithms such as the Extended Kalman Filters (EKF) [1] with submapping [2] and FastSLAM [3][4] have been used to solve this problem. Perhaps the main difference to previous work is that this algorithm is based on using data structures and optimization algorithms. There has been similar work in the past with submapping [2][3][5] (and many others) and the use of relative location [6][7].

Imagine a group of 3D point landmarks being observed from a stationary viewpoint with observational noise. Computing the average (x,y,z) location from the viewpoint over many iterations will yield an accurate map. If the viewpoint is moving, the average location calculation needs to be made invariant to the movement. It is possible to do this by assigning three landmarks to act as a basis of the calculation. One of the landmarks can be set to being (0,0,0) for translation invariance. The other two landmarks can be used to construct consistent axes from the (0,0,0) landmark for rotation invariance. The map created is now relative to these basis points.

As the viewpoint moves, landmarks leave the view and new ones appear. Landmarks that are observed in the same time interval are grouped together. The first group consists of landmarks viewed in the first iteration. Every other group consists of previously grouped and newly observed landmarks. The first group is aligned to the global map by performing a Least Square Fitting (LSF) [8] between the landmarks relative locations and their first observations. This assumes that the viewpoint in the first iteration was at location (0,0,0). Every other group has LSF performed using the previously mapped landmarks global location's to their relative location in the group.

Much of the advantages of the Relative Point algorithm come from its philosophy of retaining past untransformed observations. The previous iteration's untransformed landmarks are used in landmark registrations. Errors in current position, perhaps due to dynamic landmarks do not affect the registration process. After sufficient iterations, a landmarks grouping is revaluated to maximize its calculation interval. It is possible to change groupings and to recompute past iterations since the past untransformed observations are available.

During the regrouping process, there are enough iterations available to identify dynamic landmarks. Pair wise comparisons can be made using the past untransformed observations. Landmark pairs with a low standard deviation of distance to each other over a time interval can be binned together. An approximation can be made that only one landmark per bin needs to be compared against another bin. Two heuristics are used: if one bin has a majority of landmarks the algorithm can be stopped, and to prioritize bins with past positive comparisons. The dynamic detection algorithm is O(nlogn), even if a high percentage of landmarks are uncorrelated.

Landmarks are grouped together with other landmarks that are observed at the same time. A group is created a few iterations after a landmark is first seen. These first groupings are revaluated later to create groups with larger calculation intervals, and to perform dynamic point detection. Local relative maps are computed for each group, which are then joined with LSF to form a global map. Current position is recalculated every iteration by comparing currently seen landmarks to their global positions using LSF. Current position is used locally for matching landmarks due to backtracking and globally for closing the loops. It is possible to completely avoid the use of current position by instead relying on landmark merging. The Relative Point algorithm only recomputes landmarks seen in the current observation and is shown to be $O(n_a \log n_a)$ where n_a is the average quantity of observed points. The total quantity of landmarks

does not affect the execution speed. The Relative Point algorithm is shown to have a quick base speed, operating at a few milliseconds with a hundred average observed landmarks. It is also shown to have comparable accuracy to a 6D no odometry EKF, and is able to identify dynamic points.

II. Description

The Relative Point algorithm consists of many object oriented classes. These classes are described in the four different phases of the algorithm: registering points, computing groups, creating groups, and global matching.

A. Registering Points

The *Point* class represents an observation of an untransformed point. It consists of a (x,y,z) location and a time variable describing which iteration the point is observed in.

The *RltPoint* class consists of a circular array of *Point*s. Its main responsibility is to enumerate *Point*s given a time interval.

The *RltPointCharting* class charts a reference to every *RltPoint* seen in a given iteration. Its main responsibility is to enumerate a list of *RltPoint* references given a time interval. It uses a hash table for duplicate detection. It is used in both point matching and group creation.

The *OVLPQuadTree* is an overlapping quadtree that performs spatial subdivision for point matching. Each cell in the quadtree overlaps adjacent cells by the maximum point matching bound. This allows the point matching to only have to query one cell. In experimentation, it is shown to be faster than a kd-tree and octree depending on the point density.

Point matching begins by initialization the *OVLPQuadTree* with the past few iterations of *Points*. Then *RltPointCharting* is enumerated for the past few iterations. For every *RltPoint* enumerated, its most recent *Point* is obtained and added to the *OVLPQuadTree.* After the initialization, every new untransformed point from the current iteration is compared to *Points* in the *OVLPQuadTree.* If an untransformed point matches, it is added to the matching *RltPoint*. Otherwise global matching is used to see if there is a match. Global matching is described later. If there is still not a match, a new *RltPoint* is created. The matching *RltPoint* is then added to *RltPointCharting* for the next iteration.

B. Map Creation

Each *RltPoint* can belong to multiple groups. The *RltGroupRef* class is used to store a reference to each group a *RltPoint* is in.

The *RltLSF* class performs the Least Square Fitting (LSF) [8] algorithm between two point clouds. It returns a matrix that can transform the location of a point from one point cloud to the other.

The *RltGroup* class stores a reference of every *RltPoint* in the group. It is used to compute the *RltPoi*nt's relative and global locations.

When a *Point* is added to a *RltPoint,* the *RltPoint* notifies all of its *RltGroup*s using the *RltGroupRef*. When notified, the *RltGroup* adds itself to the list of *RltGroups* to be recomputed in the current iteration. Before it does the computation, the *RltGroup* first checks to see if every *RltPoint* has a new observation in a given iteration. If one *RltPoint* does not, it exits.

The architecture of the Relative Point algorithm is designed so that the grouping decisions and dynamic point detection is done at a level above the *RltGroup.* If there is a change, a new *RltGroup* is created. This allows a *RltGroup* to use the same three basis points to form the transformation matrix that makes the relative map translation and rotation invariant. For a new iteration, first the three basis points are used to compute the transformation matrix. The basis is created by setting the first basis point to being at location (0,0,0), the second one is placed on the negative x axis and the third is placed on the xz plane.

After the basis transform is computed, each untransformed point for a given iteration, for each *RltPoint* is transformed to relative coordinates. The relative location is added to the *RltPoint*'s average relative location. If the *RltGroup* is created in the first iteration, every *RltPoi*nt's relative location versus its first observation is used for LSF. If the *RltGroup* is created with previously grouped points, every previously grouped *RltPoint's* relative location versus its known global location is used for LSF. The matrix returned by the *RltLSF* class is used to compute the global location of *RltPoints* that have not been previously mapped in other groups.

C. Group Creation

The *RltInterval* class stores the time intervals in which a *RltPoint* is observed. It is updated every iteration a *RltPoint* is seen. Its main function is a combine function performs a union of two *RltIntervals.*

The *RltUngroupedList* class maintains a sorted by iteration list of *RltPoints.* It is used to keep track of *RltPoints* that have not been grouped yet. This avoids an enumeration over all *RltPoints*.

The *RltMapper* class performs much of the high level functionality of the algorithm. It is the entry point of the algorithm and performs the group creation.

When a *RltPoint* is first created, it is added to a *RltUngroupedList.* This list is queried in *RltMapper* when checking for *RltPoints* that have not been grouped yet. When a *RltPoint* has sufficient iterations, the group creation algorithm attempts to create a *RltGroup* to put it in. That *RltPoint* is referred to as the creation *RltPoint*.

The group creation algorithm starts by obtaining the iteration that the creation *RltPoint* is first observed. It then queries *RltPointCharting* to create two lists. One list has *RltPoints* that are already grouped and present in the interval of the creation *RltPoint*. The other list has *RltPoints* that have not been grouped yet, and present in the interval of the creation *RltPoint.* Both lists are sorted by order of the greatest quantity of iterations in the interval of the creation *RltPoint.*

Next, *RltPoints* are selected from both lists to form a group. It is unlikely that any of these *RltPoints* are present in the complete interval of the creation *RltPoint.* A compromise of a reduced calculation interval versus number of *RltPoint*s has to be reached. Either constants can be used to reduce the interval or a heuristic can be used. The heuristic is to add the three best previous *RltPoints* and the creation *RltPoint* to the *RltGroup* and combine their intervals together. Then by order of best interval, *RltPoint*s from the sorted lists are tested to see if they would improve the group. The test consists of combining the interval of a *RltPoint* to the *RltGroup's* interval. If quantity given by the combined interval size multiplied by the number of *RltPoints* increases, the *RltPoint* can be added to the *RltGroup*. *RltPoints* from the sorted lists are tested until one *RltPoint* fails the interval test.

Before the *RltPoints* placement in the *RltGroup* can be finalized, dynamic detection is performed. Each *RltPoint* is placed in a bin. Using a hash table, the bin stores every *RltPoint* the bin has already been compared to, to avoid duplicate comparisons. For every iteration of the algorithm, each bin is compare to one other bin that it has not been compared to before. Using the untransformed *poin*ts from one *RltPoint* per bin, the standard deviation of their distance to each is other calculated during the *RltGroup's* interval. If the standard deviation is less than a threshold, then the bins are merged. Otherwise their hash tables are set so the two bins are not compared to each other again.

The standard deviation threshold is selected by saving the standard deviation of the first iteration's comparisons. These saved standard deviations are sorted from lowest to highest. The threshold is selected by enumerating the sorted list until the next standard deviation is percentage wise much larger then the previous. This operation is integrated into the rest of the dynamic detection algorithm so it does not require extra comparisons.

The dynamic detection has the possibility to degrade to $O(n^2)$ if the number of uncorrelated *RltPoints* is high. Two heuristics are used that allow the detection to occur at O(nlogn). One is: that if one bin has a majority of *RltPoints,* the dynamic detection runs for one more iteration. The last iteration compares every bin that the majority bin has not been compared to, to the majority bin. Any positive comparisons are merged to the majority bin. The other heuristic is to prioritize bins that have a positive comparison in the previous iteration of the dynamic detection. The dynamic detection is shown to be O(nlogn) even with a very high quantity of uncorrelated points.

After the group is created, three *RltPoints* are chosen to be the basis of the relative map. If the three basis *RltPoints* are collinear then the basis will not be rotational invariant. Care is taken to chose *RltPoints* that are spaced far apart and non collinear.

The group creation can be run several times for each *RltPoint.* It is first run a few iterations after a *RltPoint* is first seen to initially place the *RltPoint* on the map. It can then be run a second time after more iterations have occurred. Additional grouping can be delayed up to the point where the *RltPoint* is no longer visible. The first *RltGroup* grouping checks periodically to see if there is at least one *RltPoint* in it that does not have a second grouping. If every *RltPoint* has a second grouping, that *RltGroup* is redundant and removes itself from the map.

D. Global Matching

If an untransformed landmark observation fails to match to the previous iterations untransformed observations, there is the potential that the landmark has been seen before, but not recently. This can occur when the viewpoint is backtracking or closing a loop. In this case, global matching is used. The

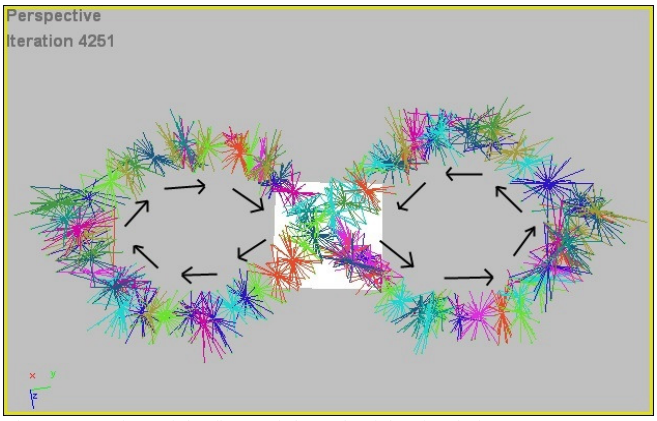

Fig. 1. Top view of the figure eight path of the simulation.

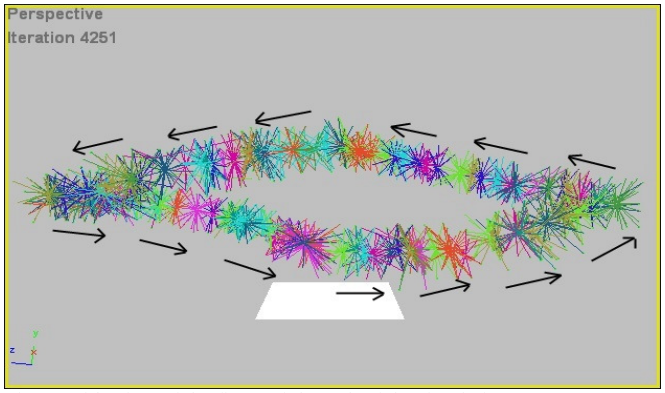

Fig. 2. Side view of the figure eight path of the simulation.

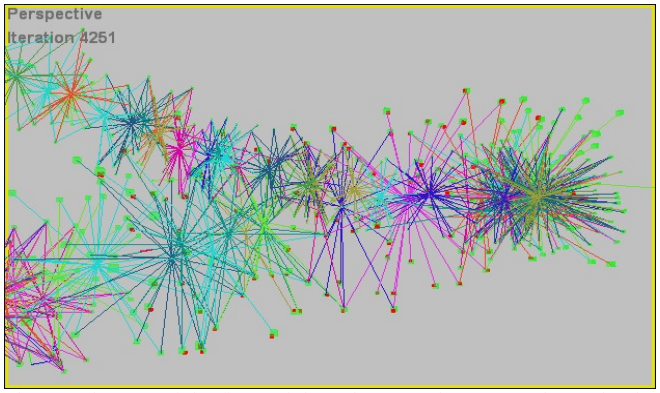

Fig. 3. Close up of the figure eight. The Green boxes are where RltPoints are mapped to, red boxes are the current observation. The lines go from the centriod of each RltGroup to every RltPoint in it.

matching uses current position. Current position is generated by performing a LSF using the current untransformed observations, compared to their global positions.

Current position is used to transform the untransformed landmark into global space. It is then matched to the global overlapping quadtree. If there is a successful match, it is added to that *RltPoint* and thus *RltPointCharting,* so the next iterations untransformed observation will automatically match to it. Note that in the case of backtracking, the current position needs only to be locally accurate as the current position is directly related to the map. However, in order to successfully close the loop, the current position needs to be globally accurate within the matching bound.

There is the possibility that if an untransformed observation has sufficient observational noise and with current position error, an untransformed observation can fail to match to its *RltPoint* global location. If this happens, a new *RltPoint* is created. After a few iterations of averaging its relative location, the new *RltPoint*'s global location should be approximately the same as the one it should have matched to. If this occurs a merging routine detects this situation and merges the two *RltPoint*s. *RltPoint*s are merged by copying over the untransformed observations, from the newer *RltPoint* to the older one. Then the older *RltPoint* is recomputed over its now larger time interval. It is possible not to use current position at all and instead rely on the point merging. This would come at a cost of higher overhead.

III. Run Time Performance

The Relative Point algorithm is tested in a simulation of a figure eight path. The results are shown in Figure 1, Figure 2 and Figure 3.

The simulation starts at the middle and travels the right loop counter clockwise with a slight incline. When it reaches the middle it is well above the starting point. It then travels the left loop clockwise with a slight decline, returning to the starting point when the left loop is complete.

Figure 4 shows the execution time for a simulation of the figure eight seen in Figure 1. Notice that the linear regression of the execution time is flat. This verifies that the Relative

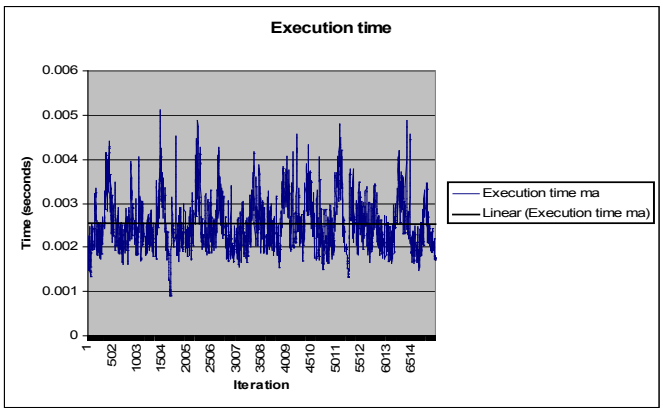

Fig. 4. Execution time of the Relative Point algorithm with approximately an average of 100 points per a given observation. The linear regression is flat denoting that the average time does not increase as more points get mapped.

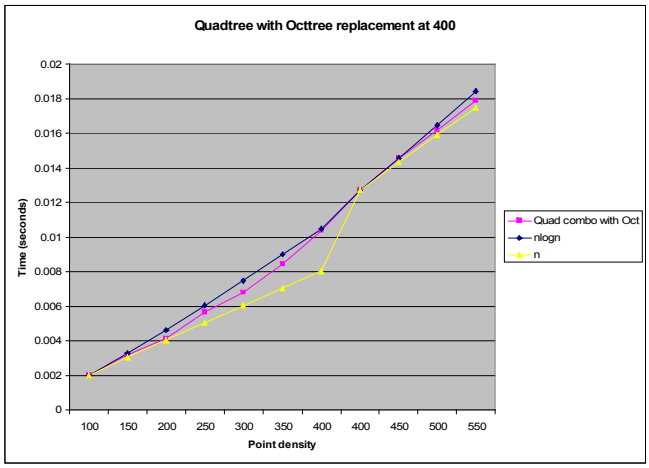

Fig. 5. Execution time with different average point densities of the figure eight. The quadtree structure degrades and become polynomial at the 400 points per 100 unit length density. It is replaced by an octree at the 400 point density. The octree has a high initialization but retains its O(nlogn) computation complexity.

Point algorithm's run time is proportional to the average quantity of points observed in a given iteration.

Figure 5 shows the execution speed of the Relative Point algorithm with increasing point densities of the figure eight simulation. The density is the average number of points per

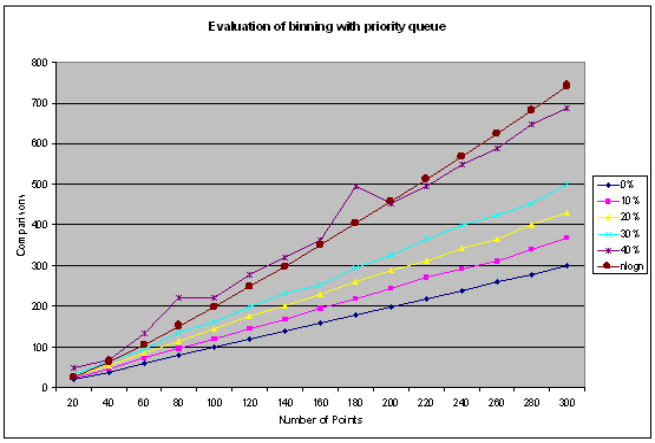

Fig. 6. Dynamic point detection with 0% to 40% points being uncorrelated. Only the 40% is near the nlogn line.

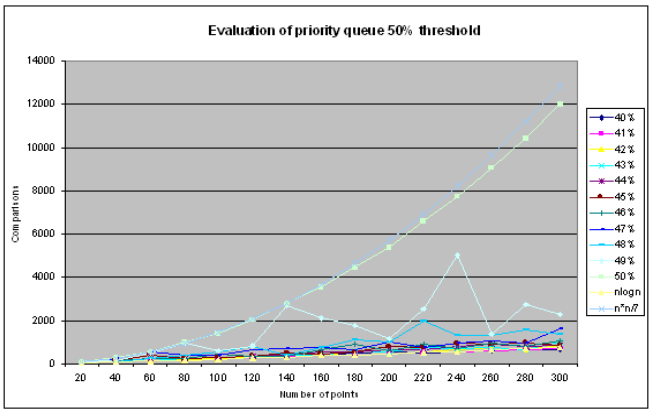

Fig. 7. Dynamic point detection with 40% to 50% points being uncorrelated. Only at about the 47% line does the algorithm start to degrade.

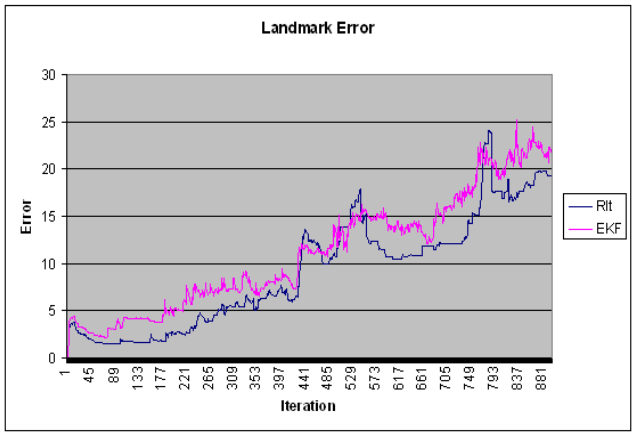

Fig. 8. Comparison of the Relative Point algorithm to the EKF.

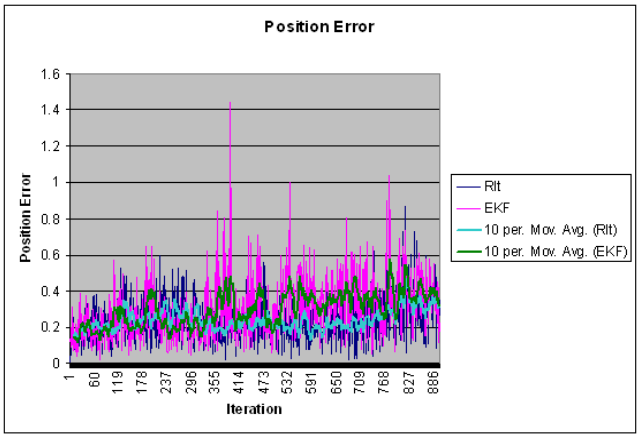

Fig. 9. Position error of the EKF versus Relative Point algorithm.

100 unit length. The viewing frustum is 70 units length but due to overlaps in turns the average density is approximately the same number as the average points per iteration. It was found that the algorithm starts to become polynomial as the point density increases. This was shown to be due to the quadtree losing its O(nlogn) execution time. At the 400 point density, the quadtree is replaced with an octree. A kd-tree is tested but is shown to be slower than both the quadtree and octree. The octree has a higher initialization but retains its O(nlogn) execution time.

The dynamic detection algorithm is evaluated for different percentages of uncorrelated points. Figure 6 shows the results of the percentage of uncorrelated points being between 0 and 40%. Notice that only the 40% line approaches O(nlogn). Figure 7 shows that the dynamic detection only starts to degrade at about the 47% line.

IV. Accuracy Comparison to EKF

The Relative Point algorithm is tested against a 6D no odometry EKF [9] obtained from the Mobile Robot Toolkit [10]. The point density is reduced to 25 points per 100 units length to allow the EKF to run. The observation noise is white Gaussian noise. Since the point density has a random distribution, reducing the density further can cause the number of observed points to go below the minimum for the Relative Point algorithm. Figure 8 shows the landmark error for 900 iterations. Unfortunately at about 900 iterations, the EKF run time was approaching 1 second per iteration and slowing down further, so the testing stopped. The average Relative Point algorithm run time was about 1 ms.

Figure 8 and Figure 9 shows that the accuracy of the Relative Point algorithm is similar to the EKF. Notice in Figure 8, that spikes in errors occur in similar places. The Relative Algorithm has sharper spikes of errors and then reductions due to the accuracy increases of the regouping process. Notice that in some places in Figure 9, the Relative Algorithm has greater location error then the EKF even though its landmark positions are more accurate for the same time iteration. This is due to the fact that the Relative Algorithm position is not cumulative. The accuracy of

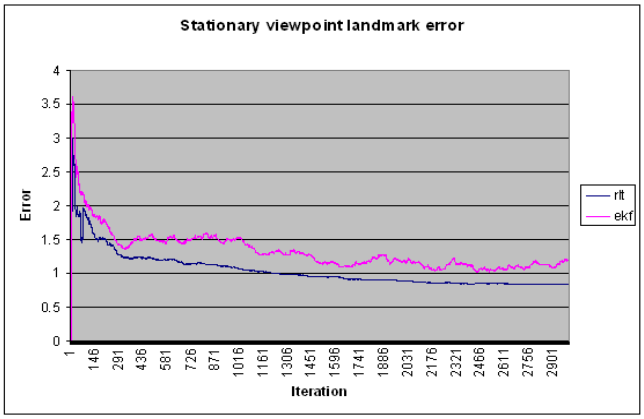

Fig. 10. Landmark error with a stationary viewpoint.

current position in the Relative Algorithm is related to the accuracy of the map, and the noise in the current observations.

Figure 10 shows the position error if the viewpoint does not move in the simulation. For both algorithms the error decreases.

Both algorithms are tested with dynamic points. Not surprisingly the EKF loses track while the Relative Point algorithm is able to identify and not use the dynamic points for the map. It is interesting to note that even before the dynamic point detection, the Relative Point algorithm is more robust than the EKF due to the use of untransformed observations for point registration.

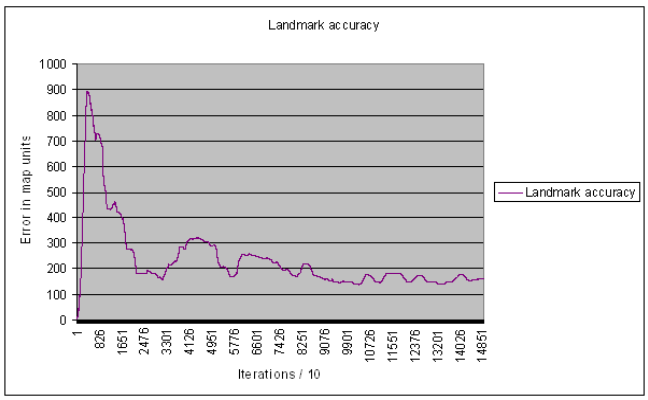

Fig. 11. Landmark accuracy of 40 passes of the figure eight.

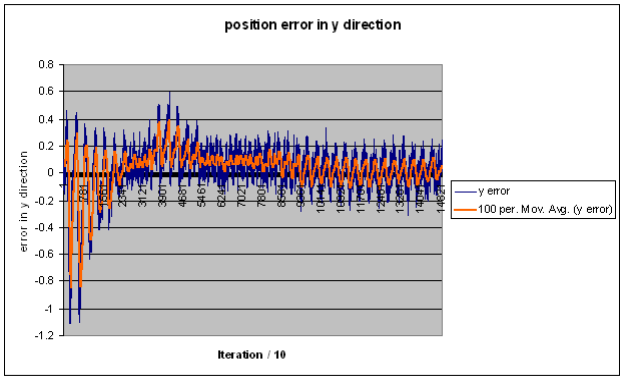

Fig. 12. Position error in y direction for 40 passes of the figure eight.

The Relative algorithm is tested with 30% of the point in the figure eight simulation being dynamic. The mapping errors are slightly higher than the simulation without noise, but the error is small enough so that the algorithm can close the loop. The increase of errors can perhaps be due to having less static points available for the map, or perhaps due to dynamic points crossing static ones and creating point registration issues.

The purpose of the comparisons is not to rank the accuracy of the algorithms versus the other. Rather it is to verify that the Relative Point algorithm has similar accuracy to an implementation of an EKF. It is interesting to note that even though the algorithms are different, the increases in landmark error appears correlated between the two algorithms.

V. Long Term Accuracy Testing

Figure 10 poses an interesting question. If the landmark error decreases after many iterations where the viewpoint is stationary, should the landmark error decreases after many loops of the figure eight?

Figure 11 shows the landmark error of 40 passes through the figure eight. The landmark error in Figure 11 decreases similarly to Figure 10, which has a stationary viewpoint.

It is interesting to look at position error in y direction shown in Figure 12. The position error in y direction is similar to the x and z graphs so they are not shown. The error is periodic and after many runs, the error seems to converge rather than decrease. It is not known why the error does not further decrease to zero. There can be an issue with bias in the simulated noise or there can be an issue with the Relative Point algorithm. Perhaps the fact that the comparisons are discretized into groups leads to the result shown in Figure 12. The error though is very small considering the path length is about 1200 units and the observational noise is a maximum of 1 unit for each direction.

VI. Conclusion

The Relative Point algorithm has a worst case computation complexity of $O(n_a \log n_a)$, where n_a is the average quantity of points visible at the same time. The computational complexity is dependent on the data structures used, in particular the structure used to do the point matching. It is important to tune the quadtree or octree so that it maintains its O(nlogn) computation complexity. The average computation time of the 550 points per 100 units figure eight simulation is 18 ms per iteration where there is an average of 537 points per observation and 8980 total points on an AMD 64 3400+. The Relative Algorithm is able to identify dynamic points in O(nlogn) time, and is shown to work with a large percentage of points being dynamic.

The accuracy of the Relative Point algorithm is compared to a 6D EKF implementation that does not use odometery. The accuracy of the Relative Point algorithm is shown to be comparable to the EKF.

The accuracy of the Relative Point algorithm is evaluated by having the robot loop through the same figure eight many times to see if the error approaches zero. After many runs, the error appears to be reduced to having a small bias. It is not known if this bias is due to the algorithm, or due to a bias in the noise of the simulated environment.

References

[1] M. Dissanayake, P. Newman, S. Clear, H. Durrant-Whyte, M. Csorba, "A Solution to the simultaneous localization and map building (SLAM) problem," *IEEE Transactions on Robotics and Automation* V 1713 Jun 2001

[2] L. M. Paz, J. D. Tardos and J. Neira, "Divide and Conquer: EKF SLAM in O(n)," *IEEE Trasnactions on Robotics.* Volume 24, No. 5, October 2008.

[3] S. Thrun, W. Burgard, D. Fox. "Probabilistic Robotics," *MIT PRESS*, Cambridge, MA, 2005.

[4] T.D. Barfoot, "Online visual motion estimation using FastSLAM with SIFT features," IROS 2005

[5] B. Lisien, D. Morales, D. Silver, G. Kantor, I. Rekleitis, H. Choset, "Hierarchical Simultaneous Localization and Mapping," *Proceedings of the 2003 IEEERSJ Intl. Conference on Intellgent Robots and System* Las Vegas, Nevada, October 2003.

[6] M. Csorba, "Simultaneous Localisation and Map Building," *PhD Thesis,* University of Oxford, 1997.

[7] P. Newman, "On the Structure and Solution of the Simultaneous Localisation and Map Building Problem," *PhD thesis*, University of Sydney, 2000

[8] K. S. Arun, T.S Huang, and S. D. Blostein, "Least square Fitting of two 3D point sets," *IEEE Transaction on Pattern Analysis and Machine Intelligence,* 9(5) 1987 pp. 698-700

[9] J. L. Blanco, "Derivation and Implementation of a Full 6D EKF-based Solution to Bearing Range SLAM," *Technical Report, Perception and Mobile Robots Research Group,* University of Malage, Spain, 2008

[10] Mobile Robot Toolkit 6D-SLAM, *http://www.mrpt.org/6D-SLAM,* retrieved January, 2011