|  |
| --- |
| Planner Module |
| SAROPS Version 2.1 |
|  |
| This document describes the basic approach and algorithms of the Planner Module in SAROPS 2.1. It also emphasizes what changes were made from the Planner Module of SAROPS 2.0.3. |
| This document contains data that is delivered to the Government with Unlimited Rights in accordance with contract HSCG44-16-F-001007 and FAR 52.227-17. |
| Thomas M Kratzke, Strategic Data Systems, 18 April 2017 |

Table of Contents

[I. Introduction 0](#_Toc478016691)

[A. Grab-a-rock and Polish-a-rock 0](#_Toc478016692)

[B. POS and SRUs 1](#_Toc478016693)

[C. Ladder Patterns, Perfect Patterns, and Terminology 1](#_Toc478016694)

[D. Optimization Problem’s Constraints 2](#_Toc478016695)

[E. SruSolution and SruSolutionArray 3](#_Toc478016696)

[F. Non-Linear Programming and Parameterization of Boxes 4](#_Toc478016697)

[G. Frozen and Floating SRUs 5](#_Toc478016698)

[H. Summary 6](#_Toc478016699)

[II. Problems with Planner 2.0.3 6](#_Toc478016700)

[A. Overlap Problem when Placing SRUs 7](#_Toc478016701)

[III. Grabbing a Rock 7](#_Toc478016702)

[A. Small Set of Particles and Large Number of Rocks 7](#_Toc478016703)

[B. Initial Placement of 1 SRU on a Probability Distribution 8](#_Toc478016704)

[IV. Bird’s Nest Algorithm 8](#_Toc478016705)

[V. Putting an SRU on Mars 10](#_Toc478016706)

[VI. Measuring Overlap 10](#_Toc478016707)

[A. Overlap Computation Algorithm Sketch 11](#_Toc478016708)

[VII. Polishing a Rock, Different Stages 13](#_Toc478016709)

[A. Preliminary Stage: Increase POS 14](#_Toc478016710)

[B. Clear Shape Constraint Violations 14](#_Toc478016711)

[C. Clear Ovl Constraint Violations 14](#_Toc478016712)

[D. Improve POS: Zero In 14](#_Toc478016713)

[VIII. Polishing a Rock, Clearing Constraints by Clearing “Worst” Constraint 14](#_Toc478016714)

[IX. Different Approximations to POS Used During Optimization 16](#_Toc478016715)

[X. How SAROPS’ Uses Planner, and LatestSeed 17](#_Toc478016716)

[XI. Planner 2.2 19](#_Toc478016717)

[A. ESS, ESS+NVG, Asymmetric Sensors, and the Use of 19](#_Toc478016718)

[B. Parameterize a Solution without a Box 19](#_Toc478016719)

[Appendix A: Polishing a Rock; the List of Minor Moves 0](#_Toc478016720)

[Appendix B: XML 5](#_Toc478016721)

[A. Root Element 5](#_Toc478016722)

[B. SRU Elements 5](#_Toc478016723)

[1. Non-Pattern SRUs 6](#_Toc478016724)

[2. Pattern SRUs 6](#_Toc478016725)

[3. Lateral Range Curve 7](#_Toc478016726)

[REFERENCES 0](#_Toc478016727)

List of Tables

[Table 1 Parameterization of Boxes by other Modules of SAROPS 5](#_Toc478016728)

[Table 2 Internal Parameterization of TS-Boxes by Planner 5](#_Toc478016729)

[Table 3 Problematic Situation when Grabbing a Rock 6](#_Toc478016730)

[Table 4 Questions Combined to Produce Planner 2.1 “Grab a Rock” Approach 8](#_Toc478016731)

[Table 5. Different PosFunctions 17](#_Toc478016732)

[Table 6 Information of a Planner Problem 18](#_Toc478016733)

[Table 7 Steps of a Planner Run 18](#_Toc478016734)

[Table 8 Uses of Planner 18](#_Toc478016735)

[Table 9 Parameterization of a Pattern in Planner 2.2 19](#_Toc478016736)

[Table 10 Root (<PATH>) Element of Planner XML 5](#_Toc478016737)

List of Figures

[Figure 1 Two Types of Ladder Pattern 2](#_Toc478016738)

[Figure 2 Optimization Problem Statement in Terms of Boxes 3](#_Toc478016739)

[Figure 3 Buffer around each leg, and Convex hull around all buffers 4](#_Toc478016740)

[Figure 4 Red Cross pair of Boxes 11](#_Toc478016741)

[Figure 5 Separating Edge 12](#_Toc478016742)

[Figure 6 No Separating Edge, and the Edge that is Closest to being a Separating Edge 12](#_Toc478016743)

[Figure 7 Reducing can be Misleading; 13](#_Toc478016744)

[Figure 8 Tangled Boxes 15](#_Toc478016745)

[Figure 9. Starting position, with “hand” at starting point of path 0](#_Toc478016746)

[Figure 10. MoveAlong and MoveNegAlong 0](#_Toc478016747)

[Figure 11. MoveAcross and MoveNegAcross 1](#_Toc478016748)

[Figure 12. IncCcwTwist and DecCcwTwist 1](#_Toc478016749)

[Figure 13. Flip and Rotate180 1](#_Toc478016750)

[Figure 14. IncAlong1 and IncAlong2 2](#_Toc478016751)

[Figure 15. IncAcross1 and IncAcross2 2](#_Toc478016752)

[Figure 16 Expand and Contract 2](#_Toc478016753)

[Figure 17. DecAlong1 and DecAlong2 3](#_Toc478016754)

[Figure 18. DecAcross1 and DecAcross2 3](#_Toc478016755)

[Figure 19. Rotate90Ccw and Rotate90Cw 3](#_Toc478016756)

Planner Module

SAROPS Version 2.1

# Introduction

In this document, we refer to the Planner module of SAROPS 2.1 as *Planner* 2.1 or simply *Planner*. Similarly, we will refer to the Planner module of SAROPS 2.0.3 as *Planner* 2.0.3, and the Simulator module of SAROPS as simply *Simulator*.

*Planner* works with the particles stored in the Particle File and produced by *Simulator*. These particles together represent the probability distribution of the state and position of the missing object. For example, some of the particles might be “life-raft particles,” some might be “disabled vessel particles,” and yet others might be “person-in-water” particles. These particles will usually be dispersed over a large region, and their positions will differ for each of *Simulator*’s times -steps. This means that the particles are moving (and changing state) and hence the probability distribution is time-dependent.

The input to *Simulator* is the information that we know about the case. For example, we might know what the pre-distress vessel was, and we might know either where the distress call came from, or what the planned route was. In addition, *Simulator* uses environmental data, shoreline data, and a bathymetric grid, to put the particles in different places at different times. In *Simulator*, random draws are used to spread the particles around, and to reflect the fact that we know neither the exact strengths of the winds and currents, nor the precise way in which they affect the missing object.

*Simulator* produces a collection of particles, and the purpose of *Planner* is to assign search assets to cover these particles as thoroughly as possible.

## Grab-a-rock and Polish-a-rock

There is no magic formula for assigning the search assets, and so *Planner* uses an algorithm to compute a reasonable assignment. It then perturbs each assignment one of several ways, and checks to see if that improves the assignment. When it cannot find any perturbations that improve the assignment, it uses the 1st algorithm to get a completely new, but still reasonable assignment, and it continues until it runs out of time.

We will use some handy terminology introduced at an Internal Program Review, that seems to communicate this very basic concept. We refer to the process of finding an initial set of boxes, as “grabbing-a-rock,” and the process of improving this set of boxes with a sequence of perturbations, as “polishing-a-rock.” Of the two parts, grabbing-a-rock is by far the more important. *Planner* 2.1 has made significant changes to both parts of the algorithm.

## POS and SRUs

The search assets are called Search and Rescue Units (or SRUs). Typically, these are helicopters, fixed wing aircraft, or vessels that can search for the missing object. The Probability of Success (or *POS*) is defined as the sum of the probabilities of the SRUs finding the particles, where the sum is taken over all the particles. Since the particles are not necessarily equally likely, we must weight them in the sum. This gives us the definition of *Overall POS*, which we simply call *POS*:

Equation 1 Definition of Overall POS

*Planner* will assign a rectangle, hereafter referred to as a “Track-Spacing box,” or “TS-box,” or simply “box,” to each SRU. A box induces a path, also known as a “pattern,” for the SRU. This pattern is used to compute the quantity in the 4th line of Equation 1. *Planner* will try to assign boxes to SRUs to maximize *POS*.

## Ladder Patterns, Perfect Patterns, and Terminology

As stated before, a box induces a pattern, and the type of pattern that it induces, is called a “ladder pattern.” A ladder pattern has an odd number of segments called “legs.” The odd-numbered legs are called “search legs,” are parallel to each other, and have the same length, which we denote as . The even numbered legs are called cross legs, are also parallel to each other, and also have the same length. We call this length the “track-spacing” and denote it by . The sequence of turns is either right, right, left, left, right, right, left, left, …” or left, left, right, right, left, left, right, right, …. The former is called “first-turn-right” and the latter is called “first-turn-left.” Examples are given in Figure 1.

Note that USCG convention has an additional classification of ladder patterns; PS and CS, but *Planner* does not use this distinction. To *Planner*, every ladder pattern is just a first-turn-right or a first-turn-left ladder pattern.

First-Turn-Right Ladder Pattern

First-Turn-Left Ladder Pattern

Figure 1 Two Types of Ladder Pattern

All lengths are in nautical miles, which we simply call “miles,” and we assume this is a 2-dimensional problem so that “parallel” means what “parallel” usually means, without being concerned about the curvature of the earth. *Planner* uses local coordinate systems to avoid significant distortion so that this approximation has negligible error. An SRU has an “effective path length” *L* which is 85% of its speed times duration-on-scene.

We use to represent the length of the side of the box that is parallel to the search legs, and to represent the length of the side of the box that is parallel to the cross legs. is always positive, but can be negative. A positive value for indicates a first-turn-right pattern, and a negative value indicates a first-turn-left pattern.

A “perfect Ts-box” is one for which the total length of the legs is , the final leg is a complete search leg, and there is a buffer of size around the pattern. All versions of *Planner* up to and including 2.1 are restricted to finding perfect TS-boxes. The restriction to perfect boxes is quite onerous; there are very few legal values of *ell* for a given *L*. Skipping the algebra, we will merely state that *ell* must be an integer divisor of *L*. One consequence of this is that when looking for “nearby” solutions, we cannot smoothly adjust *ell*. We hope to get rid of this restriction in *Planner* 2.2.

## Optimization Problem’s Constraints

There are some boxes that are not allowed, and some sets of boxes that are not allowed. For example, a box for which the induced pattern has a track-spacing that is too small, would be disallowed. We call these considerations “constraints.” Each of these constraints is expressed within *Planner* as a function that measures how much the constraint has been violated, and hence negative values for these functions are good. We will use the word “constraint” to mean either the function or the physical situation it is designed to measure.

There is a constraint for each pair of SRUs that are not allowed to overlap. Both what this means and how it is measured are different in *Planner* 2.1 than in *Planner* 2.0.3. The other constraints are that for each SRU , there is a lower bound on its track-spacing and an upper bound on the ratio . Let be the box assigned to SRU , and we’ll use (e.g.) to indicate the track-spacing of the pattern induces by the box. Summarizing and expressing the objective function and constraints in terms of boxes, we have:

Figure 2 Optimization Problem Statement in Terms of Boxes

Here, is a function that measures the overlap, and the other two constraint functions will simply be the value minus its bound. We call these constraints *Ovl* constraints, *TS* constraints, and *PS/CS* constraints respectively. The latter two are not very problematic and we group them together and call them shape constraints.

## SruSolution and SruSolutionArray

We will be a little inaccurate here. An assignment of a box to an SRU is called an “sruSolution,” and an assignment of boxes to all the SRUs is called an sruSolutionArray. Note that we can compute all the constraints for an sruSolutionArray and we say that an sruSolutionArray is “feasible” if the constraints are all at most zero. It follows that an sruSolutionArray is “infeasible” if at least one of the constraints is positive. We shall make the definition of sruSolution and sruSolutionArray more precise when we discuss “frozens” in §I.G.

In *Planner* 2.1, as opposed to *Planner* 2.0.3, “overlap” does not mean the overlap of the boxes themselves. Rather there is a constant for each SRU called the “Pattern Separation Buffer” (or ), and *Planner* 2.1’s definition of overlap requires this constant. In *Planner* 2.1, a buffer of widthis placed around each leg in a pattern, and the convex hull of the union of these buffers is called the “exclusion zone.” It is these exclusion zones that are not allowed to overlap. In Figure 3, there is a red leg surrounded by a red buffer, a green cross leg surrounded by a green buffer, and a blue search leg surrounded by a blue buffer. The yellow convex hull is the exclusion zone.

In *Planner* 2.0.3, exclusion zone was simply the TS-box. The two exclusion zones coincide if *PSB* happens to be the same as .

Figure 3 Buffer around each leg, and Convex hull around all buffers

The computations of shape constraint violations are more straightforward; these are simply the difference between the values and their bounds.

In *Planner* 2.0.3, overlap was measured by considering the areas of the two boxes, and the area of the intersection. In *Planner* 2.1, the overlap is defined as the distance one of the exclusion zones must move to eliminate overlap. The algorithm used to compute this quantity depends on the exclusion zones being convex. Boxes are already convex and this is one reason that *Planner* takes the convex hull to form its exclusion zone. Note that in Figure 3, the union of the buffers is not convex but of course the yellow convex hull is certainly convex.

This measurement of overlap allows a non-zero measure for two boxes that are clear of each other, and this is important during the polishing step In *Planner* 2.0.3, once two boxes had no intersection, the measure was necessarily zero, no matter how far apart they were. In *Planner* 2.1, we can use the fact that two boxes are far apart to affect the optimization. Other things being equal, it is better if two boxes are somewhat separated rather than being right next to each other; this provides more flexibility when moving the boxes.

## Non-Linear Programming and Parameterization of Boxes

As said before, *Planner* must find a feasibleset of boxes that maximize *POS*, and there is no closed form formula for this. The grab-a-box, polish-a-box algorithm can be considered a non-linear optimization approach, and we hope to use standard non-linear optimization libraries in *Planner* 2.2.

Optimization algorithms do not consider boxes to be variables. In an optimization algorithm, variables must be real numbers or Booleans. Hence, we must represent our problem with numbers instead of boxes, when discussing what we want the optimizer to set. There are many ways that we can use five numbers to specify a box, and we will use one of them. This recasts the problem so that now there are variables, where is the number of SRUs, instead of boxes.

SAROPS has always described boxes by using numbers. To be exact, the rest of SAROPS uses five numbers and two Booleans to specify a box and its ladder pattern as below:

1. Length of Longer Side
2. Length of Shorter Side
3. Orientation of Longer side
4. Center Latitude
5. Center Longitude
6. PS/CS (Boolean)
7. First-Turn-Right (Boolean)

Table 1 Parameterization of Boxes by other Modules of SAROPS

There are two main reasons that *Planner* does not use this parameterization: optimization algorithms do not work well with Booleans, and there is no important distinction between PS and CS patterns; they are both ladder patterns. Hence, *Planner* uses the following parameterization, which eliminates the Booleans. We have discussed and . We have 3 more variables:

1. Length of side that is parallel to search legs (must be and is called )
2. Length of side that is parallel to cross legs (is unconstrained and is called )
   1. Negative value indicates first-turn-left
3. Direction of 1st leg (0 to 360 and is called
4. Center latitude and is called *centerLat*)
5. Center longitude and is called *centerLng*)

Table 2 Internal Parameterization of TS-Boxes by *Planner*

Converting from one parameterization to another is not hard, and is done when reading the input or building reports.

From Table 2, we see that the odd-numbered search legs run in the direction of , the even-numbered ones run in the direction of , and the box is oriented so that the side of length *ell* is parallel to the 1st search leg. The cross legs all run in the direction of if and if .

## Frozen and Floating SRUs

Strictly speaking, an sruSolution is just a pattern and an exclusion zone. If a pattern is given, but no exclusion zone is, *Planner* will use *PSB* as discussed in §I.E to create one. This is useful for cases where some of the SRUs have been given patterns that the optimizer may not manipulate. We call such SRUs “frozen SRUs” or simply “frozens.” By contrast, we call SRUs that *Planner* must place, “floating SRUs” or “floaters.” *POS* is affected by frozens and *Ovl* constraints between frozens and floaters must be considered. Planner reports any *Ovl* constraint violations between two frozens, but does not consider them as constraints during the optimization. *TS* and *PS/CS* constraints do not apply to frozens.

## Summary

*Planner* addresses the problem stated in Figure 2, restricting itself to perfect boxes. *Planner* works with the quantities of Table 2 instead of the boxes of Figure 2, and uses a “grab-a-rock, polish-a-rock, repeat” approach.

In §II, we discuss general concepts about the entire optimization algorithm, especially some shortcomings discovered in *Planner* 2.0.3. In §III, we go into more detail about the grab-a-rock step. In *Planner* 2.1, there is an intermediate step between grab-a-rock and polish-a-rock, called the bird’s nest algorithm, and this is the topic of §IV. In some cases, some SRUs are often more hindrance than help, and we discuss how this occurs and how it is handled, in §V.

In §VI, we give more detail about measuring overlap, and provide the background that is necessary for discussing clearing overlap. Clearing overlap is part of the polish-a-rock step, which is discussed in §VII and §VIII.

In §IX we present a technical argument to justify not using the exact definition of *POS* during the optimization; we use it only when producing reports. In §X, we discuss how some concepts familiar to SAROPS users translate to *Planner* ideas. In §XI.A, we discuss the asymmetric sensors ESS and ESS+NVG, and some of the problems they cause for the current algorithm, and in §XI, we discuss the improvements we hope to put into *Planner* 2.2.

In Appendix A: Polishing a Rock; the List of Minor Moves, we give a detailed look at the nearby moves of the polish-a-rock step, and in Appendix B: XML, we provide a summary of the main input XML file for running a *Planner* case.

# Problems with Planner 2.0.3

Suppose we are in the following situation:

1. The probability distribution that has 2 distinct modes *A* and *B*
2. 2 SRUs,
3. Ideally should be on mode *A* and should be on mode *B*.
4. But the solution we start with has on *B* and on *A.*

Table 3 Problematic Situation when Grabbing a Rock

In *Planner* 2.0.3, grabbing a rock was based at least partially on an existing sruSolutionArray such as the one above. For some of the steps, *Planner* 2.0.3 would move only one SRU, say for example, the one that was contributing the least. In the above example, *Planner* 2.0.3 might remove , and put it back after adjusting the probability distribution based only on . Then repeat with , and go back and forth.

This strategy fails miserably for the case of Table 3. *Planner* will never get and switched to *A* and *B*. In fact, they’ll keep ending up very close to where they started. In general, moving a single SRU can lead to a cyclical sequence of solutions, and we call this “cycling.” One workaround used in *Planner* 2.0.3 and before was to leave a “copy” of in place, place the real , and then remove the copy. *Planner* 2.0.3 had some success with this, but it still often led to cycling, and in this case, it would never switch and .

*Planner* 2.0.3 ran into enough of this type of problem so that *Planner* 2.1 completely abandoned basing as sruSolutionArray on an existing one.

Another strategy that earlier versions of *Planner* 2.0.3 used, was that every other time it was to grab a rock, it moved all the SRUs, modifying the order in which it did so, to try to get significantly different rocks. Switching the strategy to doing this every time did not seem to hurt earlier *Planner*’s results, and it simplified the code a great deal, getting rid of the “make copy of SRU” part of the algorithm. *Planner* 2.1 takes this approach of moving all the SRUs farther and refines it, as we will discuss in §III.A.

In both *Planner* 2.0.3 and *Planner* 2.1, the SRUs are ordered and placed greedily one at a time. By this we mean that *Planner* places an SRU as if it’s the only SRU left, updates the probability distribution as per its placement, and then places the next SRU based on the updated probability distribution. It repeats this process until it has placed all the SRUs, and this entire process is what it means to grab-a-rock.

## Overlap Problem when Placing SRUs

When an SRU is placed over a high probability area, but there is still a large amount of probability in its exclusion zone, it is possible that other SRUs will be placed there and we will have a large amount of overlap to clear in the polish-a-rock step. In *Planner* 2.0.3, attempts were made to make regions unattractive by penalizing an SRU that came close to the center of SRUs that had already been placed. This logic was also removed, again simplifying the code. It was replaced with bird’s nest algorithm.

# Grabbing a Rock

The *Planner* 2.1 algorithm for grab-a-rock has some significantly new steps to address the example of Table 3, as well as cycling and “squatter’s rights.”

## Small Set of Particles and Large Number of Rocks

Every time *Planner* 2.1 grabs a rock, it orders the SRUs arbitrarily. In earlier versions of *Planner*, some heuristic order was placed on them. Heuristics that we tried included largest average coverage, biggest minimum track-spacing, etc. In *Planner* 2.1, we are trusting that running through the grab-a-rock step more often will work better than using a specific type of ordering. With a specific type of ordering, it is always possible to do exactly the wrong thing, and it is possible to do it over and over. Hence, 2.1 uses a more random approach and will examine far more rocks.

The 2nd new strategy in 2.1’s rock grabbing algorithm, is to randomly pick a *small* set of particles (currently set to 128) and base the placement algorithm only on these particles. The idea here is to pursue a larger number of promising rocks, letting randomly chosen particles guide the selection of rocks. This approach eliminates cycling. Moreover, with the arbitrary ordering of SRUs within the grab-a-rock step, and the bird’s nest algorithm, squatter’s rights has been all but eliminated.

## Initial Placement of 1 SRU on a Probability Distribution

Given a probability distribution, this section addresses how *Planner* initially places a single SRU. We call this the “accordion algorithm.” It was present in *Planner* 2.0.3, but it has been modified.

We will use as an approximation for *POS* in this step. To do this, we compute *POD* for a particle in a box by using the following formula.

Equation 2 ; Definition of *POD*

A grid is imposed on the a priori probability distribution and we start with this grid’s highest probability cell. This single cell can be considered as a array of cells. Using the formulation above, and ignoring considerations about perfect boxes and *TS* constraints, we might expand our current array to a or a array. In general, if we have an array of cells, we will consider arrays of size , , , and . Note that for each of these possibilities, there are up to 2 since we can (for example) grow either to the right or left.

Taking care not to cycle, and being sure to grow the array to something close to , we find a good array and then adjust it to a perfect box. The expansion and contraction inherent in decreasing and increasing the number of rows or columns, gives this algorithm its “accordion” name.

When we imposed a grid, we had to choose an orientation or direction for the . We have no a priori selection criterion, so we simply take , and for some randomly chosen , and choose our best result. The is part of the randomization of choosing this particular rock.

# Bird’s Nest Algorithm

At a talk at George Mason University on SAROPS, I was asked two questions back-to-back that caused me to consider a new approach, but not until I was driving home that night. The questions followed a part of my talk that dealt with handling overlap, and some of the difficulties I faced when dealing with it. I had been asked these two questions before, but not so close together. The questions were:

1. By placing the SRUs, wouldn’t that take care of overlap since an SRU will make the region it covers so unattractive to subsequent SRUs?
2. Can you just combine the SRUs and treat them as one SRU, and then split them up?

Table 4 Questions Combined to Produce Planner 2.1 “Grab a Rock” Approach

The answer to the first question is “definitely not necessarily.” If the first SRU is weak, there will still be plenty of probability in the region that it is covering, and to just let other SRUs go after it, would invite a very long polishing stage to clear the overlap.

The answer to the second question is “what if you *want* them split?” By combining them, you are forcing them to visit the same region.

But taken together, these questions led to the approach we are using in 2.1. First, we do not worry about overlapping SRUs when placing them sequentially; this is following the suggestion inherent in the 1st question. Again, this simplifies code, which is always a positive thing.

Then the 2nd question comes into play; not *before* the sequential placement of the SRUs, but *after* it. The collection of SRUs that *are* illegally overlapping will *then* be treated as a single SRU. We call such a collection of boxes a “bird’s nest,” and the algorithm that clears the overlap within a bird’s nest, the “bird’s nest algorithm.” This latter algorithm is very much in line with the suggestion inherent in the 2nd question. In general, the boxes comprising a bird’s nest are not lined up or orderly in any predictable way. Hence, *Planner* takes the convex hull of these boxes’ union, and splits that up in an orderly, non-overlapping parallel fashion among the constituent SRUs.

The resulting boxes are parallel, which is an attractive feature of the solution. They are close together which has benefits and problems. The problem most often encountered is that some of the boxes are too small for the *TS* constraint. Another is that they are not necessarily perfect boxes and still must be adjusted, thereby possibly introducing overlap.

To address these problems, we align the search legs with the longer side of each box and, when we adjust the box to make it a perfect box, we decrease .

Now consider a bird’s nest and assume that there are no frozens in it. After picking a direction to line up the SRUs within the birds next, the convex hull is split among the SRUs, giving those SRUs with larger more of the polygon. We pick a few (currently 2) axes of the convex hull and use the one that produces the highest *POS*.

There are problems with the bird’s nest algorithm, and most of them occur with frozens. If a frozen is part of the birds’ nest, it cannot be shifted around to occupy its place in the convex hull. Instead, *Planner* will shift the other SRUs around it, but the algorithm that *Planner* uses for this is not very efficient, especially if there are multiple frozen SRUs.

The 2nd problem with the bird’s nest algorithm is the use of the convex hull; it is bigger than the union of the SRUs’ boxes in the bird’s nest. It is conceivable that two bird’s nests will grow into each other when taking the convex hull. This has not occurred in testing, but there is code in place that addresses this. A more promising approach for this might be in *Planner* 2.2, where boxes can be shrunk almost without limit.

The final problem with this approach is a problem with the entire algorithm, and its dependence on using the entire search path. Suppose there are frozens “surrounding” an area that has a lot of probability. Two or more floaters might be attracted to that high probability area, only to have no room to move or get clear of each other and the frozens. Essentially, the floaters are trapped by the frozens.

The polishing algorithm of §VIII will address this final point. The approach there is crude and can result in far from optimal answers, but it will always find a feasible solution.

# Putting an SRU on Mars

One of the problems that occurred several times in previous versions of *Planner* was that an SRU might have no particles with any probability to look for. When this happens, no box is more attractive than any other, and the optimization algorithm of . For example, suppose an SRU *sru* can see only particles of type *OT*, but that SRUs already in place have thoroughly searched the only areas where these particles might be. Such SRUs might be within the *Simulator* run in the form of completed searches, or they might be SRUs within this *Planner* run that were placed before *sru*. *Planner* has no guidance on where to place the SRU, and the algorithm used to grab a rock can break down. This problem only gets worse since *Planner* is using a small sample for grabbing a rock.

*Planner* 2.1 addresses by introducing the concept of putting an SRU “on Mars.” When there are too few particles to place an SRU, it is classified as being “on Mars,” and it will contribute nothing to either POS or *Ovl* constraints. A special flag is put into the resulting XML file so that SAROPS can tell the user that the SRU is not being used, but in addition it will be assigned a box that is well outside of the search area. This is so the rest of SAROPS can display something for that SRU.

Note that it is possible for one rock to place an SRU on Mars, and a subsequent rock to use it. Being on Mars is a characteristic not of the SRU, but of a particular solution.

# Measuring Overlap

*Planner* 2.0.3 measured overlap by computing boxes’ intersections. For multiple boxes intersecting each other, penalties were attached to intersections of 3 or more boxes. This approach has been completely discarded in *Planner* 2.1.

One shortcoming of *Planner* 2.0.3 was that there were combinations that no small move could improve. One such pair is the “Red Cross” illustrated in Figure 4. In this example, no small move of either box can reduce the area of intersection and if this determines the measure of overlap, no polishing can be done to achieve a feasible solution.

Figure 4 Red Cross pair of Boxes

*Planner* 2.1 does not compute intersections, much less their areas. Moreover, it only considers a pair of SRUs when computing the violation of an overlap constraint. Consider a pair of SRUs that are not allowed to overlap. Because of frozen SRUs, we will consider the exclusion regions to be polygons and not simply boxes, and we will assume that these polygons are convex. This is a required part of the input.

The key quantity that Planner computes is “how far does one of the SRU’s exclusion polygons have to move to clear the overlap?” If there is no overlap, the quantity becomes the negative of “how far do we have to move one of the polygons to *create* an overlap?”

## Overlap Computation Algorithm Sketch

This section has some detail on the algorithm, but is not an important part of the document. The concept that *Planner* uses relies heavily on both polygons being convex and counter-clockwise. Hence, *Planner* will turn around any clockwise exclusion polygon and take the convex hull of an exclusion polygon that is not convex.

Suppose that one of the polygons has an edge that is lying on the x axis, that this edge starts at . Because the polygon is counter-clockwise, the other end of this edge is for some , and the other endpoints of the polygon all have non-negative *y*-coordinates. If the other polygon’s points all have negative *y*-coordinates, the two polygons are separate; one is completely above the *x-*axis, and one is completely below it. In this case, we say that the *x-*axis *separates* the two convex polygons.

If either of the polygons has an edge for which this is true, then they are separate. It is a short leap to conclude that if, for some edge *e*, we can change coordinates so that *e* starts at the origin and lies on the *x-*axis, and the other polygon lies below the *x*-axis, then they are separate. In this case, *e* is called a “separating edge” and illustrate that in Figure 5

*e*

Positive *y*-axis

Positive *x*-axis

Figure 5 Separating Edge

In this figure, we have drawn the axes relative to the separating edge *e*. The heavy arrow on *e* emphasizes that the pentagon must be counter clockwise. Note that the rectangle has no separating edge, and this illustrates that when we look for a separating edge, we must consider the edges from both polygons.

The converse is, not surprisingly, true; if we can*not* find such an edge, then the polygons *do* overlap, and we can use this to compute how far we must move one of the polygons. Simply find the edge for which the worst point on the opposite polygon is as small as possible. Here the minimum is taken over all edges of both polygons, and the (inside) maximum is taken over all points of the opposite polygon.

*e*

Figure 6 No Separating Edge, and the Edge that is Closest to being a Separating Edge

If *Planner* finds a separating edge, thereby discovering that there is no overlap it computes the green distance shown in Figure 5, and takes its negative as the violation measure. Otherwise, the red distance in Figure 6 was computed while looking for a separating edge, and *Planner* uses that.

As an aside, this algorithm depends heavily on the two polygons being convex. In mathematics, the term “separating hyperplane” is often used to describe this situation. Here the hyperplane is the line of the edge, and this line “separates” two convex sets. I have only seen the phrase “separating hyperplane” in conjunction with convex sets (see <https://en.wikipedia.org/wiki/Hyperplane_separation_theorem>).

Although it is not really misleading to consider these polygons as 2-dimensional, and the separating hyperplane as a 1-dimensional line, it is interesting to note that there’s an extra dimension in both the polygons and separating hyperplane. Because we are on a sphere, the separating hyperplane is a great circle, which is the intersection between a plane that goes through the center of the earth, and the surface of the earth. This is a 2-dimensional object. The polygons’ vertices are points on a sphere, and so they are 3-dimensional objects. The hyperplane is the 2-dimensional plane , where is the normal to the great circle.

# Polishing a Rock, Different Stages

Thus far, we have discussed grab-a-rock, but polish-a-rock has changed a great deal as well. The intent was to waste less time polishing rocks that do not look promising, while doing a segmented and orderly polishing. *Planner* 2.0.3 focused on clearing the constraints *and* improving *POS* together. *Planner* 2.1 clears the constraints and then improves *POS* in separate stages.

We will discuss the hardest constraint *Ovl* constraint soon. We mention it here to justify our use of “normalized” constraints within the optimization problem. Consider a pair of boxes corresponding to SRUs as in Figure 7, A.

Note that reducing the green box is allowed and would result in Figure 7, B. This has reduced the distance the green box must move to clear *Ovl* (as indicated by the two arrows; the 2nd one is slightly shorter) but the polygons appear to be more badly overlapped; the red one now completely surrounds the green one.

A: Starting Position

B: After Reduction

Figure 7 Reducing can be Misleading;

Hence, for optimization, a *normalized* measure is used. We divide the distance to move by the maximum distance between two points within the same polygon. The “raw” distance is used when reporting, and this is reported only if it is positive. In fact, the rest of SAROPS has little interest in the exact measure of the violation; it just needs some positive number to indicate there is overlap. Always reporting 1.0 would work just as well.

Though less critical, we note here that *Planner* works with normalized versions of the shape constraints as well.

We now discuss the four stages of polishing a rock.

## Preliminary Stage: Increase *POS*

The Preliminary Stage is intended to increase *POS* as much as possible, and we’ll clear constraints in subsequent stages. Nearby sets of boxes are examined, for example a box with a slightly different twist, or a box with a slightly perturbed center latitude. Or even one with a different value of *ell* (recall that such a value is not necessarily close to the old value of *ell*). If *POS* goes up, we make the move.

## Clear Shape Constraint Violations

The 2nd stage is clearing the Shape constraints. This is very easy because these constraints can be cleared with simple adjustments to a single SRU.

## Clear Ovl Constraint Violations

Either of the first two Stages could introduce an *Ovl* constraint, and the 3rd stage is to clear these. This is one of the most difficult and time-consuming stages. Here, we do *not* allow a move that would introduce a shape constraint violation. Because moving or expanding a box can impact other *Ovl* constraints, this one is harder and can consume much of *Planner*’s allotted time. In fact, if while clearing the constraint, the *POS* is not close to what the best-known *POS*, *Planner* will abandon this rock and go grab another one.

## Improve *POS*: Zero In

The final stage is called “zero-in” and making a move here is done simply to improve *POS*. To get to this stage, the solution must be feasible, and no move within this stage is considered if it introduces constraint violation. In this stage, we are more careful about accurate evaluation of *POS* (see §IX).

# Polishing a Rock, Clearing Constraints by Clearing “Worst” Constraint

The ideas of this section are used only when clearing *Ovl* constraints. In this section, the “worst constraint” is not some fixed constraint. As we try different solutions, the “worst” constraint will switch to being the constraint that *currently* has the highest normalized violation. Hence, we left out the article “the” that might be in front of “Worst” from this section’s title.

Whenever an sruSolutionArray is constructed as a solution, its constraint violations are computed, and they are sorted by decreasing violation. Note that if and only if the first constraint’s violation in this list is not positive, this sruSolutionArray is feasible. Our goal is to reduce the 1st constraint’s violation to zero or a negative number. If no move can do that, then we will make the 1st one no worse, and reduce the 2nd one. If we can’t do that, we will keep the values for the worst and second worst, and reduce the 3rd one, etc.

When we say “reduce the 1st one, we mean “reduce the one that is (possibly a new worst) worst.” To give an example of this concept, suppose that we have only *Ovl* constraints, and there is one for each pair of SRUs in SRUs . There are *Ovl* constraints , , and . Furthermore, suppose that their current violations have values and . If we make a move that reduces to 4, but raises to 5.5, then our “worst” has gone from 5 to 5.5, and not from 5 to 4. That’s clearly a step in the wrong direction.

*A*

*B*

*C*

Figure 8 Tangled Boxes

For a more concrete example, consider the red, green, and blue boxes in Figure 8, and suppose that the constraint violations are and . We say the “violation vector” is the ordered , and it is this that we are reducing. Here ordering is done lexicographically.

For simplicity, assume that no reduction or expansion is allowed, and no twists or movements other than left-right are allowed. We cannot reduce the worst value (which is 1), but we can easily reduce the 2nd value by moving the red box to the left or the blue box to the right. For either of these moves, the new violation vector would be something like , and this is lexicographically smaller than .

Note that if we move the blue box either direction, we will *increase* the worst. The red or green moves of the previous paragraph leave the worst the same and improves the 2nd worst. We call this type of improvement a “level 1” improvement. A level 0 improvement is one in which we improve the worst, and level *k* improvements are defined analogously.

*Planner* will pick a minimum level improvement and, among those, it will take the one with the best *POS*.

It is not hard to see that if there are no frozens, there *is* some for which there is a level-*k* improvement, no matter how small the moves are. However, the presence of frozens can make it impossible to find any improvement with a small move. *Planner* addresses this issue by considering larger and larger moves when it cannot find a small move that produces some improvement.

Again consider Figure 8, and assume that all we can do is move left or right. This time, assume that only the green (middle) box can move. Small moves cannot improve the violation vector, and we will be forced to consider larger and larger moves. Eventually we will be considering a move that throws the green box completely to the right of the blue box or to the left of the red box. This will probably all but destroy *POS*, but at least we will have a feasible solution!

A few remarks are in order here. 1. It can happen that we can only improve a constraint that is already zero or negative. The improving of a non-violated *Ovl* constraint is done to make room to improve a violated constraint. 2. Focusing on the worst constraint is a common approach in optimization. For example, in linear programming, if a problem is to maximize the minimum of several quantities , an auxiliary variable is added and for each *i*, a constraint s added. Then the objective function is simply . By focusing on reducing the worst constraint to 0, we reduce all the constraints to 0. Our use of “worst” is like the use of *z* in the linear programming problem. This proved to be far more effective and simpler than focusing on something like the sum of the positive constraints.

# Different Approximations to POS Used During Optimization

Using Equation 1 is time-consuming and there is no real need. If we examine Equation 1 we note that it is simply the expected value of the particles’ *POS* value, where the average is taken over all particles. As such, we can *estimate* this value by taking a small sample of the particles and this sample’s set of *POS* values. If we also estimate the standard deviation of the entire population by using this sample, and this estimate is sufficiently small, then we would be confident that the mean of the sample approximates the mean of the population well enough. If the standard deviation is too large, we take additional sample points. This is not the result we report, but it does result in the objective function we evaluate during the optimization. To keep objective function evaluations consistent, we start with a long list of sample points and simply work our way farther down this list when trying to reduce the standard deviation.

This idea of using a different function rather than *POS* occurs often. In fact, *Planner* uses four such functions. We call a “posFunction” a function that takes an sruSolutionArray and produces an approximation to *POS*, and the constraint violations. All posFunctions compute the constraints the same way, but the approximation (which we still call *POS*) for *POS* is computed differently by the different posFunctions. Only one posFunction computes the actual *POS* of Equation 1 and it is used only when generating reports.

It may seem odd that we are optimizing *POS* but not evaluating it, and stranger still if we were to say that we don’t really care how close our approximations are. But the only important consideration is if these evaluations guide the algorithm to a good sruSolutionArray. For the most part, they do, with the one exception being our current implementation of NoFlyThrough when dealing with ESS and ESS+NVG sensors. Even there, the resulting sruSolutionArrays are usually very good.

To describe the different posFunctions, we need another term here. A particle is “selected” if it is not landed, or we are considering landed particles, and its object-type at the end of the search period is such that at least one SRU can see it. Table 5 lists the different PosFunctions.

1. NoFlyThrough: Use , consider all selected particles.
2. SampleFlyThrough: Use derived paths, but work down a fixed list of Particles until the variance of the evaluated *POS* values for the particles is sufficiently small*.*
   1. Uses only selected particles.
3. CompleteFlyThroughSelected: Like SampleFlyThrough, but uses all selected particles.
4. CompleteFlyThrough: As with above, but use all particles, selected or not.

Table 5. Different PosFunctions

We use the relatively fast NoFlyThrough posFunction during the creation of the initial solution, and the elimination of constraints. We use SampleFlyThrough whenever we wish to see if we have a new “best solution” and when we’re in “Improve *POS*: Zero In.” We will have more to say about NoFlyThrough concerning ESS and NVG+ESS, in §XI.A. *Planner* uses CompleteFlyThroughSelected and CompleteFlyThrough only when it is generating reports.

# How SAROPS’ Uses Planner, and LatestSeed

SAROPS uses *Planner* by giving it an XML file (See §Appendix B: XML). This file provides planner with the following information.

1. Where the particle file is.
   1. *Planner* reads the particle file to find out the particles’ positions and states
2. A list of SRUs, some frozen, and some that *Planner* is to place
3. For each SRU, frozen or not:
   1. A Pattern Separation Buffer (in nautical miles)
   2. A Lateral Range Curve, also known as “LRC,” for each object type
      1. If some object type has no LRC, *Planner* uses a “blind one”
   3. What other SRUs it may overlap with
4. In addition, for each frozen SRU:
   1. A pattern, either explicitly or by supplying a TS-box
   2. An exclusion zone, in one of several ways:
      1. Explicitly with a set of points
      2. Asking *Planner* to infer a TS-box from the pattern
      3. Supplying *Planner* with a TS-box
   3. If a TS-box is used to communicate the above, then there must also be search duration and search speed
5. For each SRU that *Planner* is to place:
   1. Search speed and duration
   2. (Optional) 1st rock to use and polish
6. Limits on how long to grab and polish rocks
7. (Optional) Random seed to start with

Table 6 Information of a *Planner* Problem

The basic steps of a *Planner* run are:

1. *Planner* reads the problem’s data in by reading the XML and the particle file
   1. Note that no environmental files or shoreline data is necessary since *Planner* deals only with the particles.
2. *Planner* runs its algorithm working around the frozens and placing floaters.
   1. This is a no-op if there are no floaters.
   2. Otherwise, this terminates when one of the limits given in 6 above expires.
3. *Planner* generates a collection of reports in the form of XML files.
   1. Reformatted by other modules of SAROPS to provide html tables for user

Table 7 Steps of a *Planner* Run

SAROPS uses *Planner* to:

1. Get an initial solution (called *GetInitial*)
2. Optimize existing solutions (called *Optimize*)
3. Evaluate known solutions (called *Eval*)

Table 8 Uses of *Planner*

For *Planner*, they are all the same. For *GetInitial*, SAROPS simply does not provide any initial rocks and imposes very tight limits on the limits mentioned in Table 6. For *Optimize*, SAROPS provides as frozens, the patterns that have been decided upon, and gives *Planner* information about the floaters. It also imposes looser limits on the processing time. For *Eval*, all SRUs are frozen, and the limits are irrelevant.

SAROPS allows the user to repeatedly optimize, hoping for a better answer. In previous versions of *Planner*, the new rocks were based somewhat on the old rocks. So SAROPS would do repeated *Optimize* runs, providing *Planner*’s latest solution as a starting point and Planner could continue from where it left off.

In *Planner* 2.1, the rocks are always chosen based on a random number generator (*RNG*), and if that *RNG* is initialized with the same seed for each of these *Optimize* runs, *Planner* will simply re-generate the same solutions every time. Hence, SAROPS must supply *Planner* with a different *RNG* so that *Planner* will explore different rocks. In addition, much of *Planner*’s algorithm is based on comparing its current solution with the best one that it has found. Therefore, SAROPS must also provide *Planner* with the best solution that *Planner* has come up with in previous *Optimize* runs, so that *Planner* has something to compare its current solutions with.

Hene, for subsequent *Optimize* runs, SAROPS provides *Planner* with a new *RNG* by giving it a random number seed, called *LatestSeed*. If there is a *LatestSeed* in the XML, *Planner* will use that as its first seed. Otherwise, it uses the constant stored in Sim.properties. Moreover, *Planner* is constantly generating new seeds based on its current *RNG* and it returns in its output files a seed that SAROPS can use as an *RNG* input to *Planner* for its next *Optimize* run.

An alternative way of using *Planner* would be to take the time spent in multiple *Optimize* runs, and simply give *Planner* that much time for one *Optimize* run. This is clearly the most efficient use of *Planner* and the goal of the *LatestSeed* mechanism outlined in this section, is to be as few rocks behind a single *Optimize* run as possible.

# Planner 2.2

## ESS, ESS+NVG, Asymmetric Sensors, and the Use of

In the bird’s nest algorithms of §III.B and §IV, we use a much faster computation of *POS* than Equation 1. In these algorithms, *Planner* uses the traditional calculation. Since a TS-box is not as good an approximation when detection is down-creep-only, there are larger inaccuracies here, and *Planner* 2.1 has not addressed them. For example, the initial TS-box of a single SRU covering a square stationary rectangle will be centered on the square, whereas it should be offset since particles are only detected on one side.

In our parlance, the rock that we are grabbing is a good rock, but it will take some polishing to move it over to its correctly offset position. *Planner* 2.2 will correct this problem.

## Parameterize a Solution without a Box

The biggest change though, will be the addition of a sixth variable to indicate how much of the available effective path length the pattern will use. In fact, the parameterization of a solution will change dramatically, and will be based on the pattern rather than the box. In *Planner* 2.1, we compute a box and derive the pattern. In *Planner* 2.2, we will compute a pattern, and derive the box. The pattern will be parameterized by a small set of numbers, and the box will fit around the pattern.

1. Start point Latitude
2. Start point Longitude
3. Direction of 1st Leg
4. Length of 1st Leg
5. Length of 2nd Leg (positive or negative to indicate first-turn-right or first-turn-left)
6. Amount of path used

Table 9 Parameterization of a Pattern in *Planner* 2.2

# Appendix A: Polishing a Rock; the List of Minor Moves

We call the moves to nearby solutions “minor moves.” When we say “nearby,” we mean that the corresponding sruSolutionArrays are very similar. In fact, all but one of them will be the same, and the one that is different will differ only by a slightly different orientation, or a slightly different center point, or some other such minor adjustment.

In Figure 9 through Figure 19, we list the minor moves. Assume that we start as in Figure 9. Each succeeding figure shows the result of a small move.

In Figure 9; the “hand” indicates where the path starts.

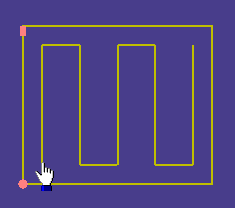


Figure 9. Starting position, with “hand” at starting point of path

Now we show the possible adjustments, together with the name of the adjustments by superimposing (in cyan (i.e., light blue)) the adjusted version of the dark-yellow rectangle, and setting the “hand” on the starting point of the adjusted rectangle.

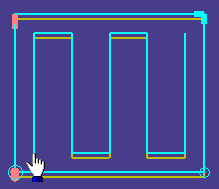
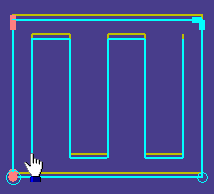
 

Figure 10. MoveAlong and MoveNegAlong

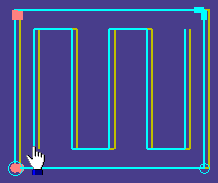
 

Figure 11. MoveAcross and MoveNegAcross

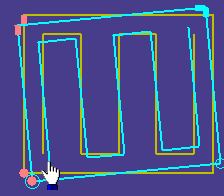
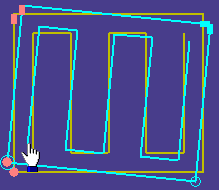
 

Figure 12. IncCcwTwist and DecCcwTwist

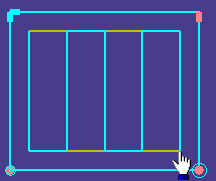
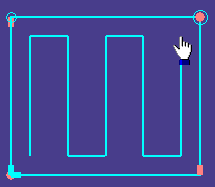
 

Figure 13. Flip and Rotate180

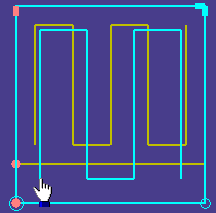
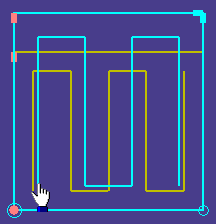
 

Figure 14. IncAlong1 and IncAlong2

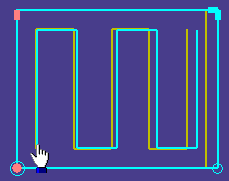
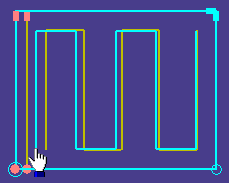
 

Figure 15. IncAcross1 and IncAcross2

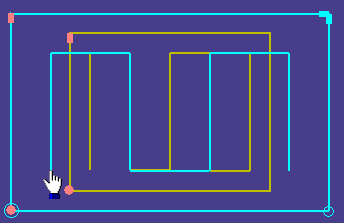
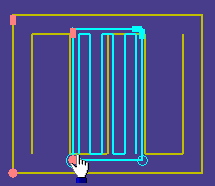
 

Figure 16 Expand and Contract

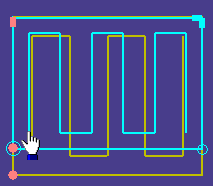
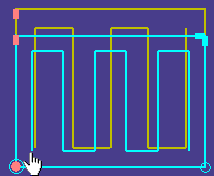
 

Figure 17. DecAlong1 and DecAlong2

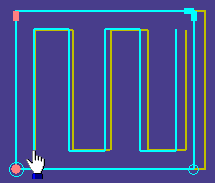
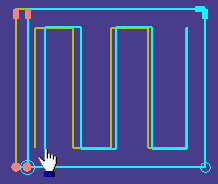
 

Figure 18. DecAcross1 and DecAcross2

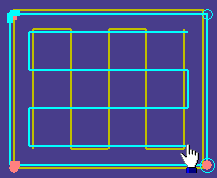
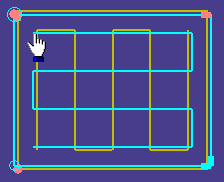
 

Figure 19. Rotate90Ccw and Rotate90Cw

The adjustments in Figure 10 and Figure 11 are four ways of moving the center. Instead of just increasing or decreasing the latitude or longitude of the center, which would introduce overlap an unnecessary proportion of the time, we “slide” the rectangles along their sides. These “slides” are of length ¼ nautical mile.

The twists of Figure 12 are 1/64 of a full revolution. The 1/64 is a hard-coded constant.

The adjustments of Figure 14 and decrease the number of search legs by 1, but keep the width of the rectangle constant. IncAlong1 pushes the rectangle one way and IncAlong2 pushes it the other. We do this to again avoid unnecessarily introducing overlap, when the rectangles are “packed” together. The adjustments of Figure 17 increase the number of search legs by 1.

The moves of Figure 15 and Figure 18 increase the track spacing by a constant factor, and keep the number of search legs the same, as well as the value for *ell*. The constant factor is a hard-coded constant, and is set to 17/16.

The expand and contract adjustments decrease and increase the number of legs by 1, but keep the search leg length the same.

Note that the path, when we do a Rotate180, appears the same if we have an odd number of search legs; we simply start at the other end. The example above has this characteristic. *Flip* would have the same path if we had an even number of search legs and again, we would simply be starting at the other end.

Rotate90Ccw and Rotate90Cw make a (usually) small change to the rectangle to accommodate the path flying the opposite direction.

# Appendix B: XML

The XML input is much simpler for *Planner* than it is for *Simulator*. There are no environmental files or shoreline files. The XML simply states where the particle file is, and the specifications of the SRUs. In addition to the particles, the particle file also has a comment that contains the XML for the *Simulator* run that created the particle file.

In the rest of this section, when referring to an attribute name or a tag name in the XML, we use a mono-spaced font.

## Root Element

The root element of the *Planner* XML has the tag PLAN and the following attributes:

1. particleFile: Full path of the particle file
2. searchForLandedParticles (Boolean) Whether landed particles count towards the POS when optimizing
3. searchForAdriftParticles (Boolean) similar to above
4. plannerTimeInSeconds: Number of seconds for *Planner*
5. maxNJumpsWithNoImprovement:
   1. No units
   2. Number of times *Planner* is to “Grab a Rock” without getting an improved POS value, before it gives up
6. latestSeed:
   1. No Units
   2. Seed for Planner to use in its algorithm

Table 10 Root (<PATH>) Element of *Planner* XML

## SRU Elements

The only elements that are direct sub-elements of the root element are SRU elements. There is one of these for each SRU in the problem (whether frozen or floater), and its tag is SRU.

Every SRU element has attributes that provide the *PSB* value (separationBuffer, see §I.D units are NM), id (id, String), name (name, String), and on Mars status (onMars, see §V, optional, Boolean) attributes. The id and name attributes together form a unique identifier. There are four types of SRUs; *blank, pattern, box,* or *frozen-box.* A *box* SRU will have a sub-element with tag <BOX>, a *frozen-box* will have a sub-element with tag <FROZEN\_BOX>, and a *pattern* SRU will have a sub-element with tag <PATTERN>. A *blank* SRU will have no <BOX>, <FROZEN\_BOX>, or <PATTERN> sub-element.

Currently, there are no *frozen-box* SRUs. Planner must place *blank* and *box* SRUs, whereas *frozen\_box* and *pattern* SRUs are frozen.

### Non-Pattern SRUs

We consider all but *pattern* SRUs first. Each SRU that is not a *pattern* SRU must have the following attributes on the SRU element: commence-search-time (cst, value is a dtg), search duration (duration, units are mins), speed (speed, units are kts), and minimum track spacing (minimumTrackSpacing, units are NM). If it is a *box* or a *frozen-box*, there will be a sub-element with tag <BOX> or <FROZEN\_BOX> and that tag will have attributes specifying the parameters as in Table 1. These are the length (length units are NM), width (width, units are NM), center point latitude (centerPointLat, no units), center point longitude (centerPointLng no units), first-turn-right (firstTurnRight, Boolean), PS or CS (pathType, String PS or CS), and orientation (orientation, units are degrees clockwise from north). In addition, the cst and duration attributes are repeated here.

### Pattern SRUs

*Pattern* SRUs are the most complicated. These SRUs represent SRUs for which the user knows exactly what the SRU’s pattern is, and simply wants *Planner* to plan the other SRUs around it. However, this *pattern* might have come from a TS-Box and therefore the user knows what he wants *Planner* to use for a creep direction and a track-spacing. In this case, the user will add additional attributes specifying these quantities to the <PATTERN> sub-element, and they are creepDirection, (units are degrees clockwise from north) and trackSpacing (units are NM). If these are not given, *Planner* will attempt to derive them from the collection of waypoints and if it cannot, it will assume that the pattern does not originate from a TS-box.

*Pattern* SRUs’ <PATTERN> sub-elements specify the pattern with a sequence of <WAYPOINT> sub-elements. Each of these has attributes defining the latitude (lat, no units) longitude (lng, no units) and time of the waypoint (dtg, standard date-time-group format).

*Pattern* SRUs have an additional optional sub-element to specify the exclusion zone. Again, if the pattern comes from a TS-box, and the user wants to specify it, the user will give a <BOX> sub-element to do this. This is *not* the exclusion zone, but rather the TS-box; *Planner* will use that box and *PSB* to derive the exclusion zone.

Another option is for the user to supply the exclusion zone directly. He can do that with an <AREA> sub-element. If this is present, the <AREA> element will have sub-elements, and each of these will have attributes providing the latitude and longitude of the points of the exclusion zone. Note that the actual tag of these sub-elements is irrelevant, and that the <AREA> element can have no sub-elements that do not specify points of the exclusion zone.

If no specification for the exclusion zone is given, *Planner* will derive one as in §I.E.

### Lateral Range Curve

Each SRU will have a lateral range curve for each object type that it can look for. If a lateral range curve is not given, then *Planner* assumes that the SRU cannot be used at all for this object type. In addition, the attribute <isActive> can be supplied to indicate that this SRU is supposed to ignore this object type during optimization, but report its effect during some of the final reports. For each object type that this SRU can be used for, the <SRU> element has a sub-element with tag <COMP\_OBJECT\_TYPE>. Within this, there is the specification of the lateral range curve and the optional attribute isActive.

# REFERENCES

1. Computer Aided Search and Rescue Planning (CASP), Volume 1
2. Computer Aided Search and Rescue Planning (CASP), Volume 2
3. National Search and Rescue Manual, Volume I: National Search and Rescue System, Joint Chiefs of Staff, U.S. Coast Guard, Joint Pub 3 50, COMDTINST M16120.5a, 1 February 1991
4. Addendum to The United States National Search and Rescue Supplement (NSS)  
   To The International Aeronautical And Maritime Search And Rescue Manual (IAMAR),  
   U.S. Department Of Homeland Security,  
   U.S. Coast Guard, COMDTINST M16130.2d, 29 April 2004
5. Memorandum “SAROPS Lateral Range Curves,” USCG (G-Opr-1), 16131, 11 March 2004
6. Enclosure (1) To Memorandum “SAROPS Lateral Range Curves,” USCG (G-Opr-1), 16131, 11 March 2004, “Discussion of SAROPS LRC And MSPP Issues”
7. Enclosure (2) To Memorandum “SAROPS Lateral Range Curves,” USCG (G-Opr-1), 16131, 11 March 2004, Excerpt From “Elements, Modules, And Algorithms For Planning Optimal Searches
8. Software Requirements Specification For The Naval Search And Rescue (NAVSAR), Naval Oceanographic Office, NAVSAR 01-12, December 2001