|  |
| --- |
| VS/SS Optimization Using Revised Rounding |
| SAROPS Version 2.2 |
|  |
| *This document describes the basic approach and algorithms for optimizing VS and SS patterns in the Planner Module in SAROPS 2.2.* |
| 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, 21 March, 2019 |

Table of Contents

[I. Introduction 0](#_Toc5193353)

[A. Terminology 0](#_Toc5193354)

[II. Rounding: SpecBox to TsBox Mappings 2](#_Toc5193355)

[*A.* *VS* 2](#_Toc5193356)

[B. *LP* and *SS* 2](#_Toc5193357)

[1. SpecToTs 3](#_Toc5193358)

[2. TsToSpec 3](#_Toc5193359)

[C. SpecToTs and TsToSpec 4](#_Toc5193360)

[1. SecToTs 4](#_Toc5193361)

[2. TsToSpec 5](#_Toc5193362)

[III. Stages of Optimization 5](#_Toc5193363)

[A. Initial Placement 5](#_Toc5193364)

[1. VS 5](#_Toc5193365)

[2. SS 6](#_Toc5193366)

[B. Birds’ Nest 6](#_Toc5193367)

[C. Minor Moves 6](#_Toc5193368)

[IV. REFERENCES 6](#_Toc5193369)

List of Tables

[Table 1: Decision Variables for *Planner* 0](#_Toc5193370)

[Table 2: Basic Terminology 1](#_Toc5193371)

[Table 3: Additional Terminology 1](#_Toc5193372)

[Table 4: Identities for *N*, *Ts*, and *SS*/*LP* 4](#_Toc5193373)

List of Figures

**No table of figures entries found.**

VS/SS Optimization Using Revised Rounding

SAROPS Version 2.2

# Introduction

In this note, we describe the optimization algorithms for *VS* (i.e., Sector Search) and *SS* (i.e., Expanding Square) patterns. Until now, we have optimized only *LP* (i.e., “Ladder Pattern,” or “Area Search” or *PS*/*CS*) patterns. The basic concept behind the problem is very similar to the optimization of *LP.*

A pattern, whether it is a *VS*, *SS*, or an *LP*, is specified by a small set of numbers, and it is this small set of numbers that *Planner* manipulates as it searches for a good solution. This set of parameters is:

1. Where the rectangle is (lat/lng of center)
2. Orientation
3. For *SS* or *LP*, the one or two dimensions of the rectangle

Table 1: Decision Variables for *Planner*

We use the term “Box” for “Rectangle.”

For *SS* patterns, there is 1 dimension specified in item 3 of Table 1, and there are 2 dimensions for *LP*. For *VS*, there are *no* dimensions to specify; a *VS* pattern is completely specified by Table 1, items 1 and 2, and so we discuss *VS* very little in this section.

## Terminology

We will use the following terms in this discussion.

1. For an *LP*, *Along* means "Length for a *PS* and Width for a *CS*."
2. For an *LP*, *Across* means "Width for a *PS* and Length for a *CS*."
   1. For an *SS*, length equals width, and so *Along* and *Across* both refer to length.
3. *Ts* means *track-spacing.*
   1. For an *LP* pattern, this is the length of the 2nd leg.
   2. For a *VS* pattern or an *SS* pattern, this is the length of the 1st leg.
   3. For an *LP*, *Sll* (search leg length) means the length of the odd numbered legs.
4. For either *LP* or SS, *TtlLegLenPlusTS* means “The sum of the legs' lengths plus 1 *Ts*.”
5. *Epl* = 0.85 \* speed \* duration (effective path length)
6. An *LP* pattern is *perfect* if and only if:
   1. Has an odd number of legs
   2. The turns are , , -, -, , , -, -, … where is either 90 or -90
   3. The odd numbered legs have the same length
   4. The even numbered legs have the same length
7. An *SS* pattern is perfect if and only if:
   1. Has an odd number of legs
   2. The turns are all either 90 or all -90
   3. Each even numbered leg as well as the last leg, is the same length as its predecessor
   4. The odd numbered legs except for the last leg, is the same length as its predecessor plus the length of the 1st leg
8. A *VS* pattern is perfect if and only if:
   1. It has 9 legs
   2. The turns are all 120 or all -120
   3. The legs all have the same length

Table 2: Basic Terminology

We also have the following terms.

1. A box *wraps* a perfect *SS* pattern or a perfect *LP* pattern if and only if:
   1. Its sides are parallel to the legs of the pattern
   2. It has a buffer of *Ts*/2 around the legs of the pattern
2. A box *wraps* a perfect *VS* pattern if and only if:
   1. Its sides are of length 3*Ts*
   2. One of its sides is parallel to the 1st leg
3. *tsInc* (track spacing increment) is a constant and for the foreseeable future, *tsInc*=0.1.
4. A *rounded* *pattern* is one for which both *Ts* and, for an *LP*, *Sll*, are integer multiples of *tsInc*.
   1. A *rounded box* is a box that *wraps* a *rounded pattern*.
5. For an *LP* pattern:
   1. The *search legs* are the odd numbered legs
   2. The *cross legs* are the even numbered legs
   3. We use the expression *Nsl* for “number of search legs”
6. For an *SS* pattern, the “number of *half-laps*” is denoted by *Nhl*, and defined by:
   1. If the pattern is a single point, we say *Nhl*=1.
   2. For any other *SS* pattern, we define *Nhl*=(*NLegs*+1)÷2.

Table 3: Additional Terminology

It may be surprising that in item 6 of Table 3, a single point is considered to be 1 *half-lap*; it “looks like” 0 *half-laps*. Moreover, 2 *half-laps* “look” more like 1 *half-lap*, etc. However, when we move from *n half-laps* to *n*+1 *half-lap*s, we “add a track-spacing to the last segment, and then two more segments, each of length *Nhl*×*Ts*.” Hence, it is consistent to think of a single point as a *half-lap*; add *Ts* to its (zero-length) last segment, and then two more segments each of length 1×*Ts*.

Furthermore, the identity *n*2 +2*n*+1=(*n*+1)2also suggests our convention. When we go from 1 *half-lap* to 2, we should add 2(1)*Ts*, then 1×*Ts* to get to 2 half-laps. For these and other reasons, we define *Nhl* as above.

In this note, we assume that, for a given *Pattern-Variable*, the *Epl* is a constant. In particular, we are not taking into account transits. Note that if we were, “*Epl*” would simply refer to the *Epl* allocated to the pattern and so most of what we say here would be easily generalized.

# Rounding: SpecBox to TsBox Mappings

Starting in Sim/Plan 2019.3, we changed how we round our patterns. This change affected *LP* patterns as well as *VS* and *SS* problems, and so we discuss all three pattern kinds here.

For *LP* and *SS* problems, we must deal with the following challenge:

For a fixed Epl and input specifications (Along,Across), we must find a perfect rounded pattern for which TtlLegLenPlusTS is at most EPL, but as close to EPL as possible, and the perfect rounded box wrapping this pattern is as close as possible to (Along,Across).

Equation 1: Fundamental Rounding Problem

The input specification (*Along,Across*) is called the *SpecBox* and the *perfect rounded* box is called the *derived TsBox* or simply the *TsBox*. Note that the *wrapping* boxes of items 1 and 2 of Table 3 are the *TsBox*es. Also, the problem is to convert the dimensions from a *SpecBox* to the dimensions of a *TsBox*; this has nothing to do with either the placement or the orientation of the pattern (items 1 and 2 of Table 1);

## *VS*

For *VS*, there are no dimensions to convert to a *TsBox*. We *do* have a rounding constraint, but that rounding constraint, and the requirement that we use as much of *Epl;* as possible, greatly simplifies the issue. For *VS* patterns, instead of *TtlLegLenPlusTS*≤*EPL*, we simply have a closed form formula for *Ts*; *Ts*=⌊*Epl*/(9×*tsInc*)⌋×*tsInc*. This gives us the maximum *Ts* for which the *VS* pattern is at most *Epl*. Assuming that *tsInc*=0.1, this translates to “round down to a multiple of 0.9” and divide by 9 to get *Ts*.

## *LP* and *SS*

The rest of this section concerns deriving a *TsBox* from a *SpecBox* for both *LP* and *SS* patterns. The optimizer always works with unrounded *SpecBox*es and so must always convert *SpecBox*es to *TsBox*es to create patterns for evaluating *POS*. Prior to 2019.3, *Planner* created unrounded *TsBox*es from *SpecBox*es during the optimization, and then rounded the winning unrounded *TsBox*es at the end of the run. Although the *TsBox*es were not rounded during the optimization, they were perfect and had *TtlLegLenPlusTs* exactly equal to *Epl*.

### SpecToTs

Starting with 2019.3, *Planner* forms *rounded* *TsBox*es from unrounded *SpecBox*es during the optimization, by using a function that we call *SpecToTs*. For any *SpecBox* *Spec*, *SpecToTs(Spec)* is the rounded *TSBox* corresponding to *Spec*. The optimization value corresponding to *Spec* is obtained by evaluating the path within *SpecToTs(Spec)*.

Because *Epl* is a constant, perhaps our notation should be *SpecToTs(Epl,Along,Across)*, but we suppress the 1st argument in this discussion to avoid excessive notation. This should cause little confusion. Also, for *SS* patterns, there is only one dimension of interest so we will use *SpecToTs*(*Along*) or *SpecToTs*(*Across*) or *SpecToTs*(*Along*,*Across*) interchangeably for *SS* problems

The problem is actually more complicated than what has been laid out here, because of how the rest of SAROPS uses *SpecToTs*. There are two uses of *SpecToTS* outside of *Planner*. For the 1st one, they gather an arbitrary Spec *(Along,Across)*, an *Epl*, and call *SpecToTs*. This is what *SpecToTs* was written to do, and causes no extra problems.

#### Idempotency of SpecToTs

But the rest of SAROPS also calls *SpecToTs* with a *TsBox* and the original *Epl* and this *is* problematic. Despite the fact that *Epl* can possibly (probably) have shrunk when going to the *TsBox*, the original *Epl* is given. For this reason, it is important that *SpecToTs*∘*SpecToTs*=*SpecToTs*, where the “∘” means “function composition” and the “=” is interpreted as “equal functions.” The mathematical term for a function that doesn’t change when composed with itself is *idempotent*; we are requiring *SpecToTs* to be idempotent.

The idempotency of *SpecToTs* is difficult to prove for the function we are using. We simulated thousands of boxes with a wide range of *[Epl,Along,Across]* values (always assuming that *Ts*=1) and found no example that demonstrated that our function was not idempotent. Still, it would be better if the rest of SAROPS used the *SpecBox*es when they use the original *Epl*, adjust the *Epl* downward if necessary, or at least include an argument that identifies it as a result of some previous call to *SpecToTs* (with this *Epl*).

### TsToSpec

2019.3 uses another algorithm that we call *TsToSpec*. *SpecToTs* and *TsToSpec* are not inverses of each other since *SpecToTs* is not one-to-one. It is true however, that for any *SpecBox* *SpecBox*, *SpecToTs*∘*TsToSpec∘SpecToTs(SpecBox)*=*SpecToTs*(*SpecBox*). We use this fact when we “warm-start” an optimization with a *TsBox*. We apply *TsToSpec* to get a *SpecBox* that we can continue with in the optimization. When we apply *SpecToTs* to this new *SpecBox*, we get the same *TsBox* as before so this is an appropriate *SpecBox* to continue the optimization iterations.

Since *SpecToTs* is supposed to be idempotent, this is not completely necessary. However, lacking a proof of the idempotency, and the infrequency with which *TsToSpec* is called, we elected to apply *TsToSpec* for the warm-starts.

# SpecToTs and TsToSpec

In this section, we examine *SpecToTs* and *TsToSpec* more closely and give a rough outline of how these functions are computed.

There is a strikingly similar role that *Nsl* and *Nhl* play for the *LP* and *SS* patterns. In fact, we use the notation *N* for *Nsl* for an *LP* problem and *Nhl* for an *SS* problem, and provide the context only when necessary.

We start with the following identities for a perfect rounded box based on the same *Epl:*

1. *N*=*Across*/*Ts*=*Epl*/*Along*
2. If *Along*=*Across* (as is always the case for *SS*), then *N*2×*Ts*=*Epl*

Table 4: Identities for *N*, *Ts*, and *SS*/*LP*

## SpecToTs

*SpecToTs* is very similar for *LP* and *SS*. An overarching feature of these computations is that, because *Ts* must be a multiple of *tsInc*, we are really solving for the integer *nTsIncs*. In addition, for an *LP* problem, we must solve for the integer *nSllIncs*, where *nTsIncs*×*tsInc*=*Ts* and *nSllIncs*×*tsInc*=*Sll*. In addition, we must solve for the integer *N*.

For both *LP* and *SS*, we start with the obvious 2 candidates *N*, based on *N*=*Epl*/*Along*; simply round *Epl*/*Along* up and down. Then we set *nTsIncs* to *max*(1,⌈ [*Across*/*N*]/*tsInc* ⌉), and *Ts*=*nTsIncs*×*tsInc.*

We must ensure that we do not have a zero-length track. For *LP* we do this by insisting that *nSllIncs*≥1 and *N*≥1. To “leave room for *nSllIncs* to be at least 1, we may have to decrease *nTsIncs*. For *SS*, we have no *nSllIncs* to worry about, but we must insist that *N*≥2 since if *N*=1, corresponds to a track with no length. Other parts of *Planner*’s computations assume that every pattern has at least 1 non-vacuous segment. In fact, *SpecToTs* starts by eliminating nuisance cases in which *Epl* is too small for either *Along* or *Across*. We omit the details of this part of the algorithm.

We then take advantage of our establishing the values of both *N* and *nTsIncs* to compute *nSllIncs* for an *LP* problem. We use *Epl*=(*Sll*+*Ts*)×*N* to compute the value for *Sll* that would use *Epl* exactly, and round down to the nearest multiple of *tsInc*.

Hence, we have a solution for each of the 2 possible values of *N*. Whether it is an *LP* or *SS* problem, we have (*N*0,*Ts*0) and (*N*1,*Ts*1). For the *LP* problem, we also have *Sll*0 and *Sll*1. For *LP*, we have two possible boxes; one for (*N*0,*Ts*0,*Sll*0) and one for (*N*1,*Ts*1,*Sll*1). We compare the two resulting rectangles to the original *SpecBox*’s (*Along,Across*), and take whichever has a smaller symmetric difference.

## TsToSpec

When *Planner* is given a starting rectangle for a *Pattern Variable* it is, unfortunately, the *TsBox* from a previous run, and not the *SpecBox*. Moreover, the *Epl* that comes with it is the *Epl* for the *SpecBox*. If *Planner* were given the *SpecBox*, it could simply continue with that *SpecBox*.

What is needed is a map from *TsBox* to *SpecBox* such that when we apply this map, the result is such that applying *SpecToTs* will produce the input *TsBox*. In this way, we can continue the optimization with a *SpecBox* that is as good as the unknown underlying one from which the *TsBox* was derived.

Fortunately, this is not too hard because of the nature of *SpecToTs*. We are not given *Ts*, but we know that it is a multiple of *tsInc* and that it divides the *TsBox*’ *Across* value.

Suppose also that the *TsBox* we were given is (*Along*,*Across*). Expressing *Across* as a multiple *M* of *tsInc*, we simply look at all of the divisors of *M* to find the possible values for *N*, where *Ts*=*N*×*tsInc*. For each of these, it is a simple matter to derive a value of *Sll\** by using the identities *Along*=*Ts*+*Sll and Epl\**=*N*×*Ts*. For each (*N*,*Ts*,*Sll*) triple, we increase *Sll* to *Sll*\* so that *N*×(*Ts*+*Sll*\*)=*Epl*. We then apply *SpecToTs* to the *SpecBox* determined by (*N*,*Ts*,*Sll*\*) to see if it results in the *TsBox* parameterized by (*N*,*Ts*,*Sll*). We claim without proof that there will be at least one such triple (*N*,*Ts*,*Sll*\*), and we have our *SpecBox*.

For *SS* problems, we again look for the divisors of *Across*. This time though, there is no *Sll* to increase to *Sll*\*. Hence, we increase *Ts* to *Ts*\* to “use up” *Epl*. Again, we state without proof that there will be at least one divisor of *Across* that is itself an integer multiple of *tsInc*, so that the resulting *SpecBox* can be converted to the given *TsBox*.

# Stages of Optimization

There are three stages of optimization: 1. Initial Placement, 2. Adjustment during the Birds’ Nest algorithm, and 3. Minor Moves. The 1st one is simple; given a probability distribution, find a good choice for the *SpecBox*, including the position, orientation and, for *LP* or *SS*, dimensions. The 2nd is like the 1st except that we are restricted to some narrow band. The 3rd stage has been discussed before in the SDD for *LP*, and we must adjust the discussion only slightly for *SS* and *VS*.

## Initial Placement

In the 1st stage of optimization, we use only the placement and size of the *TsBox* and not the pattern itself.

### VS

We have yet to state what the *TsBox* even *is* for a *VS* pattern, so we do that now. The *TsBox* of a *VS* pattern is the square that is parallel to the 1st leg, centered at the center of the pattern, and has dimension 3*Ts*. Since the size and shape of the *VS* box is not a decision that *Planner* makes, Planner simply tries different values for the center point’s latitude and longitude, as well as 6 different possible orientations. Because of the “incompressibility” of a *VS* pattern’s *TsBox*, we often refer to this as “rattling a marble around in an empty coffee can.” Because the *TsBox* is not round like a marble, perhaps we should say “rattle a single die around in a rectangular can,” but we will refer to a marble anyway.

### SS

An *SS* pattern is not as simple as a *VS* pattern, but its lack of a 2nd dimension variable makes it much simpler than an *LP* pattern. We don’t simply “rattle a marble around,” but our algorithm is pretty close to being that simple. We simply choose a few potential sizes for the *TsBox* (or the marble), and, for each one, do the same as for the *VS*. The sizes we choose are the ones that correspond to *Ts*=*MinTs* and *Ts*=*AverageSweepWidth*.

## Birds’ Nest

For both *VS* and *SS*, we actually avoid doing this. The logic behind this reluctance, is that the *VS* or *SS* is usually done 1st and, in the interest of finding the object as quickly as possible, we will leave these patterns alone. We only move one after we have tried to clear overlap by moving the *LP* patterns that we are allowed to move.

For both *VS* and *SS*, our algorithm is the same as that for the Initial Placement, except that we keep it aligned with the band that is assigned to the pattern during the Birds’ Nest refinement.

## Minor Moves

The Minor Moves are principally a subset of minor moves of an *LP*. For *VS*, we allow only the same shifts and twists that we do for the *LP* problem. For *SS*, we disallow all of the “shape changers” except for EXPAND and CONTRACT. These are the only two that retain the “squareness” of the *SpecBox*, which we require.

We *could* have had analogs for the *LP*’s “moving of one side,” but we elected not to for the sake of simplicity. So far, the optimization is working well enough with our more limited set of minor moves.

# REFERENCES