

Formal SPARK Verification of Various Resampling Methods in Particle Filters

THESIS

Osiris Terry, 2d Lt, USAF

AFIT-ENG-MS-22-M-067

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

DISTRIBUTION STATEMENT A
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views expressed in this document are those of the author and do not reflect the official policy or position of the United States Air Force, the United States Department of Defense or the United States Government. This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

AFIT-ENG-MS-22-M-067

Formal SPARK Verification of Various Resampling Methods in Particle Filters

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Electrical Engineering

Osiris Terry, B.S.C.S.

2d Lt, USAF

March 25, 2022

DISTRIBUTION STATEMENT A
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AFIT-ENG-MS-22-M-067

Formal SPARK Verification of Various Resampling Methods in Particle Filters

THESIS

Osiris Terry, B.S.C.S.
2d Lt, USAF

Committee Membership:

Kenneth M. Hopkinson, Ph.D
Chair

Clark N. Taylor, Ph.D
Member

Douglas D. Hodson, Ph.D
Member

Laura R. Humphrey, Ph.D
Sponsor

AFIT-ENG-MS-22-M-067

Abstract

This thesis makes two major contributions. First, it serves as a proof of concept for the benefit of automated software verification tools, such as AdaCore's SPARK tool suite, when building mission-critical applications. Second, the software verification in this thesis concentrates on verifying a particle filter for use in tracking and estimation, a key application area for the Air Force. The development and verification process described in this thesis is a demonstration of the power, limitation, and compromises involved in applying automated software verification tools to critical embedded software applications.

The critical role that particle filters play in target tracking and estimation make the reliance on their logic ever more present. The use of random variables when resampling particles inside of a particle filter makes it difficult to determine if the algorithm is consistent with every iteration. The proofs that were written and designed in this paper are used as validation to ensure that the most commonly used types of resampling methods can be trusted for an infinite number of iterations. Formally verifying software that use floating point numbers (i.e. numbers that include fractional values) can be problematic, but many real-world applications depend on floating point numbers. Particle Filters were chosen, in part, because they use an approach that allows for an integer representation of the floating point numbers, enabling us to construct proofs with a concise value. The formal verification of particle filters will provide a proof of concept for software verification tools and show how they can benefit the Air Force by ensuring the software is safe.

Table of Contents

Page

Abstract iv

List of Figures vii

I. Introduction 1

1.1 Problem Background 1
1.2 Research Objectives 2
1.3 Document Overview 2

II. Background and Literature Review 4

2.1 Particle Filters 4
2.2 Systematic Resampling 6
2.3 Stratified Resampling 8
2.4 Multinomial Resampling 12
2.5 Residual Resampling 13
2.6 Automated Software Verification 13
2.7 AdaCore SPARK Tools 14

III. Verification of Resampling Methods 17

3.1 Preamble 17
3.2 Construction of Systematic Resampling Proof 17
3.3 Formal SPARK Verification of Systematic Resampling 29
3.4 Construction of Stratified Resampling Proof 34
3.5 Formal SPARK Verification of Stratified Resampling 40
3.6 Additional Resampling Methods 46
3.6.1 Discussion of Multinomial Resampling Proof 47
3.6.2 Discussion of Residual Resampling Proof 48

IV. Application of Particle Filter 52

4.1 Preamble 52
4.2 Particle Filter Design 52
4.2.1 SatSim 52
4.2.2 SatMod 54
4.2.3 Particle_Filter 57
4.2.4 Additional Methods 59
4.3 Proof Analysis 61
4.4 Particle Filter Satellite Simulation 62

Page

V. Conclusions 69

5.1 Limitations 70
5.2 Future Work 71

Appendix A. Helper_Methods 72

Appendix A. Resample 76

Appendix A. Main Method 82

Appendix A. Python Simulation 84

List of Figures

Figure		Page
1	Particle Filter–Belief Distribition	5
2	Sampling Array	7
3	Sampling Array with weights	8
4	Loop Invariants: Failure at first iteration	15
5	Loop Invariants: Failure at arbitrary iteration	15
6	Loop Variants	16
7	Systematic Sampling Array	25
8	Minimum and Maximum for Systematic Sampling	27
9	Stratified Sampling Array	37
10	Proof Summary	61
11	True Location and Measurements of Satellite	62
12	Systematic Timesteps	64
13	Stratified Timesteps	67

I. Introduction

1.1 Problem Background

The construction of reliable software is a grueling process that takes countless hours to develop. There are numerous legacy systems that have not been verified and new mission critical systems are often created without using formal verification techniques during their design or construction. This is done due to a lack of understanding of the available tools as well as time and cost pressures. The trouble in efforts to ensure software reliability become far more difficult, time-consuming, and expensive if they are approached as an afterthought.

Target tracking and estimation is one of the fields mentioned that utilizes many legacy systems that have not been verified. Target tracking is a dynamic state estimation problem that stretches to a variety of different types of environments, such as military, air traffic control, aerospace, autonomous vehicles, robotics, and many more. The main concern of the problem is to determine the state of a target that is assumed to be a random variable by using observation of the target itself or any other variables that are associated with it. This is done through sequential bayesian inference, more specifically, with the use of a particle filter. A particle filter uses an array of weights that are called particles to approximate the bayesian prior and posterior. Each particle represents a state and the weight assigned to that particle determines our confidence of that particle being our true state [16]. Different from parametric filters, the PF particularly appeals to nonlinear systems affected by non-Gaussian

noises [1]. Since particle filters are so popularly used it stresses the importance that the implementation of the algorithm does exactly what it is supposed to do. For each individual implementation of the particle filter algorithm, the structure remains the same but the code can vary for certain functions. These changes are mainly due to the differences in the covariance matrix for each implementation. The algorithm that stays the same is the resampling portion of the particle filter. Conveniently, the resampling algorithm is the crux of the particle filter.

Proving the core method used in particle filters would be of value to any systems that utilize target tracking and estimation as it would provide a level of confidence that the particle filter is consistently and correctly doing its job. In addition, the proof would demonstrate the power of automated software verification tools.

1.2 Research Objectives

The questions this research aims to answer are:

- How automated software verification can be applied to critical Air Force problems by concentrating on particle filter estimation and control?
- What successes and pitfalls are encountered when applying software verification to filters used in estimation and control and how they demonstrate the value of automated software verification?

1.3 Document Overview

Chapter II presents background knowledge of the basic idea behind particle filters, SPARK, systematic resampling, stratified resampling, multinomial resampling, and residual resampling. Chapter III describes the methodology used to prove systematic resampling and stratified resampling. In addition, Chapter III goes into further depth

about the logic in multinomial and residual resampling and why a proof generated with the same methodology we used for systematic and stratified resampling would not generate any significant findings. Chapter IV shows the construction and results of the simulation described in Chapter III. The goal of the simulation is to compare our SPARK resampling methods to commonly used python versions of the same code. The python versions of the resampling methods include the floating point error that we get rid of in SPARK. The simulation also provides visual meaning to each of the resampling methods that we proved, along with an example of how particle filters are used. To conclude, chapter V summarizes what this thesis has accomplished and potential future follow-on research topics for resampling method proofs.

II. Background and Literature Review

This chapter provides the necessary background information to understand the experimental methodology and the results that follow. The first section will provide a high-level overview of particle filters, systematic resampling, stratified resampling, multinomial resampling, residual resampling, and SPARK–the language used to prove each of the resampling methods.

2.1 Particle Filters

Particle filters are nonparametric filters with the purpose of sampling a distribution using a fixed set of samples. The samples are then forward predicted using Eq (1) and that prediction is updated with a given measurement from the system; u(t) is the control and x(t) is the state. The idea behind particle filters is simple and, thus, there are many different implementations but all use the same basic principles.

$$\dot{x}(t) = f(x(t), u(t)) \tag{1}$$

A belief distribution, $F(x(t_i))$, is the probability that state x $\in R^n$ is any particular value at time t_i $\in$R$^+$ [1]. Using Eq (1) we can create a prediction of the belief distribution to find the value of t_{i+1} that is called the prior. Since the belief distribution is continuous it is impossible to generate a prior for every value, but if we take sample N states from the belief distribution we can generate N prior particles. The samples that we are taking from the belief distribution are called particles. The prior at t_{i+1} is used with the measurement being received at t_{i+1} to generate a new weighting for each of the particles. The particles, with their weighting, are a statistically valid representation of the posterior probability distribution. The posterior is then used to create a new belief distribution for time t_{i+2}, enabling the filter to propagate belief

idstributions over time, using both the dynamics and measurements to the system. Figure 1 is taken from [1] and shows a visual representation of what happens to the belief distribution when generating a posterior. This algorithm is extremely versatile but it comes at the cost of extremely high computational requirements for developing a posterior.

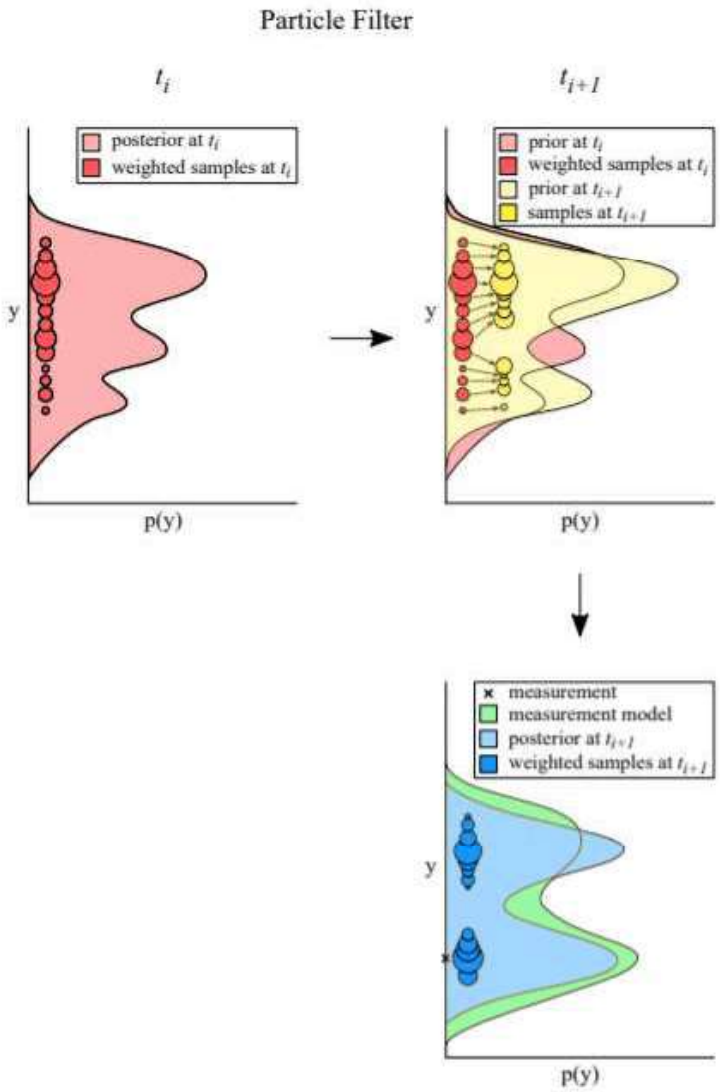


Figure 1: Particle Filter–Belief Distribution [1]

2.2 Systematic Resampling

Systematic Resampling is a type of probability sampling method in which sample members from a larger population are selected according to a random starting point but with a fixed, periodic interval [3]. This interval, called the sampling interval, is calculated by dividing the population size by the desired sample size. An example of this would be if you wanted to select a random group of 1,000 people from a population of 50,000 using systematic sampling. To do so, all the participants must be placed in a list and a random starting point needs to be selected. Once the list is formed we need to pick a Fixed Interval, the way we do this is by taking the population and dividing it by the number of samples, which comes out to a Fixed Interval of 50 (50,000 people/1,000 samples = 50 people). Now we want to add some randomness to our sampling by picking a random starting point but it is not as simple as randomly picking a starting location, we need to make an upper limit to the random start. For example, if we were to pick our random start location at the 51st person and then we sample every 50 people after this starting location, we would be sampling the 50,001st person on the list as the last sample, which does not exist. The upper limit in systematic resampling is equal to the Fixed Interval because that means that the maximum value of the last sample is equal to 50,000, which is the final person on the list [2].

The example above assumes that every person on the list is of equal importance, right now we are using a person as a unit of measurement for the amount of steps we are going to go until the next person is sampled [4]. If each person had a number assigned to them that deemed how important they were and that number was normalized so that if each person's number in the population was added up it would equal 1.0 then it would change what we defined to be our population size. Before, the population size was the sum of all the people on the list but now that people are

represented by a weight, our population size is the sum of all of the weights, which is 1.0. Now let's see how this has changed our Fixed Interval and random start: our Fixed Interval is equal to the population size divided by the number of samples we want to take and the upper limit of the random start is equal to the Fixed Interval [5, 6, 7]. Since our population size is equal to 1.0 and the number of samples that we want has not changed our Fixed Interval and upper limit of our starting location is 1/1000. This version of systematic sampling is exactly how it works in particle filters because each particle has a weight assigned to it which represents how confident we are that it is our true location. The basic formula behind systematic sample arrays in particle filters is that the Fixed Interval is equal to 1/N, N being the number of particles. For designing the systematic sampling array with code, the for loop would range from 0 to N-1 and the sampling array inside the for loop equals sampling_ array[I] = I * 1/N + Random offset, where the offset has an upper limit of 1/N. This means that the last index value of a systematic sampling array has a maximum value of N/N = 1.0.

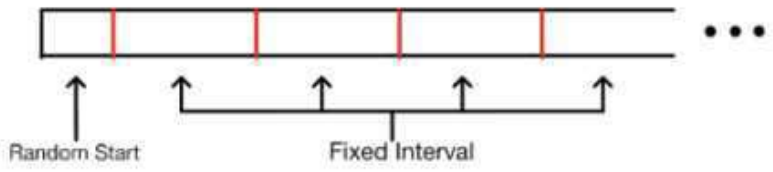


Figure 2: Sampling Array

The sampling method of systematic resampling can be represented visually, as shown in figure 2. The first block in figure 2 is the random starting point that has a maximum value of the fixed periodic interval that is shown in every block after the first one. The red line represents when a sample will be taken which means that the number of red lines and number of particles will always be the same [8, 9, 10]. The visual representation of the weights has a very similar design but instead of having a

pattern the weights are unordered and random. The only rule that the weights follow is that they must add up to 1.0. In Figure 3, the red arrows show how samples will be taken from the weights array. In code, every time a sample is taken the index of the weight is stored in an array and this array will be used to form the new particles array by telling it what particles from the old array will be copied [11]. When designing a Systematic sampling array in code we would initiate it within a for loop that ranges between 0 to the number of indexes we wanted. Inside the for loop we would have sampling_array[I] = I * (Fixed Interval) + Random offset. This would create an array that begins at the random offset and then increments by the Fixed Interval for every index.

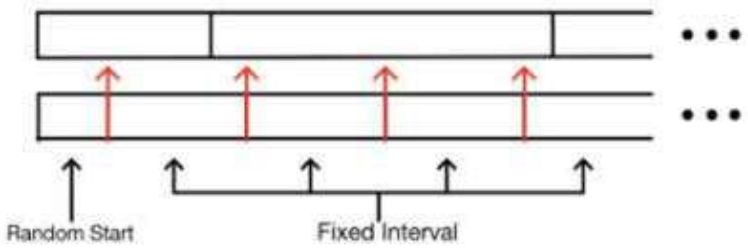


Figure 3: Sampling Array with weights

2.3 Stratified Resampling

Stratified Resampling by design is similar to Systematic Resampling [12, 13]. The difference between the two is that in Systematic Resampling the random offset in the first index that is added to every element in the array is the same value for every index it is added to, while in Stratified Resampling the random offset value changes for every index. When designing the stratified sampling array in code the change in the formula is very minimal, we still initiate it inside of a for loop ranging from 0 to the number of indexes we want but the random offset that is added is now an array. That is, the formula for the sampling array inside of the for loop is sampling_array[I] = I *

(Fixed Interval) + Random offset[I]. As stated, the change in the code is very minimal but the nature in how it samples and the rules it follows have changed significantly, the extent to how much has changed will be shown in the verification section.

Using the same example that we used in Systematic Resampling, if you wanted to select a random group of 1,000 people from a population of 50,000 using systematic sampling all the participants must be placed in a list and 1000 random starting points need to be selected [14, 15, 16, 17]. Once the list is formed we need to pick a Fixed Interval by taking the population and dividing it by the number of samples which comes out to a Fixed Interval of 50 (50,000 people/1,000 samples = 50 people). A Fixed Interval of 50 means that we will iteratively sample every 50th person on the list. Now we want to add some randomness to our sampling by adding in a random number for every iteration of our Fixed Interval. The upper limit is the same as systematic resampling. When looking at the last index of the sampling array formula (sampling_array[I] = I * (Fixed Interval) + Random offset[I]) we know that the maximum value for it is equal to 50,000 because every random number that we add to each iteration of our Fixed Interval has a upper limit of 50 people; anything greater would result in the possibility of the last index being greater than 50,000 which would be out of bounds for our list [18, 19, 20].

The example above assumes that every person on the list is of equal importance, right now we are using a person as a unit of measurement for the amount of steps we are going to go until the next person is sampled. If each person had a number assigned to them that deemed how important they were and that number was normalized so that if each person's number in the population was added up it would equal 1.0 then it would change what we defined to be our population size. Before, the population size was the sum of all the people on the list but now that people are represented by a weight, our population size is the sum of all of the weights, which is 1.0. Now lets see

how this has changed our Fixed Interval and our list of random numbers: our Fixed Interval is equal to the population size divided by the number of samples we want to take and the upper limit of the random numbers list is equal to the Fixed Interval. Since our population size is equal to 1.0 and the number of samples that we want has not changed our Fixed Interval and upper limit of our starting location is 1/50,000. This version of stratified sampling is exactly how it works in particle filters because each particle has a weight assigned to it which represents how confident we are that it is our true location. The basic formula behind stratified sampling arrays in particle filters is that the Fixed Interval is equal to 1/N, N being the number of particles. For designing the systematic sampling array with code, the for loop would range from 0 to N-1 and the sampling array inside the for loop equals sampling_ array[I] = I * 1/N + Random offset[I], where the offset has an upper limit of 1/N. This means that the upper limit of a interval in stratified resampling is 2 * Fixed Interval and that the last index value of a systematic sampling array has a maximum value of sampling_array[N-1] = (N-1) * 1/N + Random offset[N-1] = N/N = 1.0 [21, 22, 23, 24].

The major difference in the natures of stratified and systematic resampling is that we know in systematic resampling that every 50th person after the random starting location will be selected (more broadly, every iteration of the Fixed Interval after the starting location will be selected) while in stratified resampling we are limited to a range of possible samples per iteration [25, 26]. The interval in stratified resampling is derived from the Fixed Interval to become a range that goes between [Fixed Interval + min(Random offset[I]), Fixed Interval + max(Random offset[I])]. In regards to the code for stratified resampling the sampling array equals sampling_array[I] = I * (Fixed Interval) + Random offset[I] and the interval between each sample equals the difference between two samples being taken (sampling_array[I] - sampling_array[I-1]).

In systematic resampling the interval equals sampling_array[I] - sampling_array[I-1] = I * (Fixed Interval) + Offset - ((I-1) * (Fixed Interval) + Offset) = Fixed Interval. In Stratified Resampling the interval equals sampling_array[I] - sampling_array[I-1] = I * (Fixed Interval) + Random offset[I] - ((I-1) * (Fixed Interval) + Random offset[I-1]) = Fixed Interval + (Random offset[I] - Random offset[I-1]). The minimum and maximum values of this formula are found through considering the minimum and maximum values of (Random offset[I] - Random offset[I-1]). First let's establish a minimum value for our random offset, in our implementation of a particle filter we are going to use a very small decimal but not exactly zero. The only reason for the minimum not being inclusive of zero is because it allows the first sample being taken to sample a particle with a weight of zero, this does not have a major effect on the particle filter by any means but it is still a preference to not include zero [27, 28]. Now that we know our range of Random offset[I] equals (0, Fixed Interval] we can determine the range of (Random offset[I] - Random offset[I-1]) is (-Fixed Interval, Fixed Interval). Now we need to take one more step back to look at the range of the interval for stratified resampling: sampling_array[I] - sampling_array[I-1] = Fixed Interval + (Random offset[I] - Random offset[I-1]) = (0, 2 * Fixed Interval). This means that when we look at two samples, with no other information, they could be sampled at the same spot or sampled 2 * Fixed Interval apart [29, 30, 31].

The most interesting and complicated part of Stratified Resampling is that the range of the interval actually changes when given information about the previous interval. Lets say that an interval between two samples equaled the maximum (2 * Fixed Interval), the interval between the next two samples changes. The maximum value, 2 * fixed inverval, only happens when Random offset[I] is at its maximum value and Random offset[I-1] equals its minimum value. When we get to Random offset[I+1] the interval between Random offset[I+1] and Random offset[I] is equal

to (min(Random offset[I+1]) - max(Random offset[I]), max(Random offset[I+1]) - max(Random offset[I]) = (-Fixed Interval, 0) because we were given that the Random offset[I] is at its maximum value. This means that the interval for stratified resampling, given that the previous iteration was at its maximum value, is the Fixed Interval + (Random offset[I] - Random offset[I-1]) = (0, Fixed Interval). When the previous iteration is equal to the minimum value, 0, the interval between Random offset[I+1] and Random offset[I] is equal to (min(Random offset[I+1]) - min(Random offset[I]), max(Random offset[I+1]) - min(Random offset[I]) = (0, Fixed Interval) because we were given that the Random offset[I] is at its minimum value. This means that the interval for stratified resampling given that the previous iteration was at its minimum value is the Fixed Interval + (Random offset[I] - Random offset[I-1]) = (Fixed Interval, 2 * Fixed Interval). The main take away from this concept of Stratified Resampling is that previous intervals affect the range of future intervals [31, 32, 33].

2.4 Multinomial Resampling

Multinomial Resampling is unique amongst resampling methods because it is completely random [34, 35]. Every index in multinomial resampling could sample any existing particle. When designing the multinomial sampling array in code we generate as many random numbers as there are particles and each random number can range between [0,1]. In systematic resampling the random number could range from (0, Fixed Interval], but since we want our random number to be completely random for our sampling array we will increase the range for this random number so that it could sample the last–this is why the maximum for the random number is 1. In this sampling method, we allow the random number to be equal to 0–it does not matter if the method samples a particle with a weight of 0 because the objective is for it to

be as random as possible. On its own, this sampling method is rarely implemented, instead it is used as a conceptual idea when talking about other resampling methods or it is used alongside another resampling method [36, 37, 38, 39].

2.5 Residual Resampling

Residual resampling is another unique sampling method that will mathematically calculate the minimum number of samples we want of a particle based on the weight. The way this is done is by multiplying the weights of each particle by the number of particles, N, then truncating the calculated float into an integer [40, 41, 42, 43]. The calculated integer value represents the number of copies that will be made of that particle. After we calculate this value for every particle we sample the rest of the particle with another resampling method. A more in depth explanation of this resampling method will be provided in the results and analysis section [44, 45].

2.6 Automated Software Verification

Ada is a programming language built for embedded systems with low memory requirements, direct interfacing with hardware, real-time system, and low-level systems programming. Ada provides safety features that detect defects at an early stage, usually at compilation time or using static analysis tools. Spark is a subset of Ada targeted at functional specification and static verification, it comes with a set of development and verification tools for Ada [3]. These verification tools provide a level of assurance with regard to the correctness of your code this is much higher than most other languages. It is assumed throughout this paper that the reader is familiar with Ada/SPARK as a language, if additional learning is required to understand the proofs there are multiple resources that may help in the references section [4, 5, 6, 7]. It is also assumed that the reader has a basic understanding of Python, or programming

in general, as there is a simulation section in this paper that will be interfacing with our ada scripts with python; the python code is very simple and can be interpreted with very little background knowledge but if further reading is required for a specific library that is used or the language overall then there are some helpful references [8, 9, 10, 11, 12, 13]. In addition to the background readings provided, there will be a quick review on how spark works and the syntax we will be using throughout the proofs.

When mixing SPARK Code and Ada Code it is important to be able to differentiate between the two. Understanding the distinction allows for clear reading and understanding of which sections of code will compile with the SPARK restriction requirements that is required to permit formal verification. There are three basic rules that allow for a reader to differentiate between SPARK and Ada code: Ada completions of SPARK declarations are only allowed in subprograms, SPARK code cannot be wrapped around Ada Code, and a Ada subprogram body cannot contain SPARK code but it can at a library level [5]. An easy way to tell if a section is in SPARK code is by looking for any syntax that uses SPARK_Mode. This syntax allows for switching between a SPARK code section and an Ada section. There are two other important syntaxes that will be used throughout our proofs: Loop invariants and Loop variants[46, 47, 48].

2.7 AdaCore SPARK Tools

Loop Invariants are used in loops and perform two checks on a specified property to make sure that it holds for every iteration of the loop. The first check makes sure that the property holds for the first iteration of the loop and the second check makes sure that the property holds for an arbitrary iteration of the loop, assuming that the previous iteration holds. The checks on the first iteration and on an arbitrary iteration

are independent from each other[5]. Examples of some simple Loop Invariants are shown in Figures 4 and 5, along with their GNATprove outputs [49]. Figure 4 shows a case where the first iteration fails and Figure 5 shows a case where a check on an arbitrary iteration fails [50]. All of the loop variant and loop invariant figures were taken from the SPARK User Guide [4].

```
6     Prop := False;
7     for J in 1 .. 10 loop
8        pragma Loop_Invariant (Prop);
9        Prop := True;
10    end loop;
```

```
simple_loops.adb:8:30: high: loop invariant might fail in first iteration
    8 |       pragma Loop_Invariant (Prop);
      |                              ^~~~
  e.g. when Prop = False
simple_loops.adb:8:30: info: loop invariant preservation proved
```

Figure 4: Loop Invariants–Failure at first iteration [4]

```
12    Prop := True;
13    for J in 1 .. 10 loop
14       pragma Loop_Invariant (Prop);
15       Prop := False;
16    end loop;
```

```
simple_loops.adb:14:30: info: loop invariant initialization proved

simple_loops.adb:14:30: medium: loop invariant might not be preserved by an arbitrary_
↪iteration
   14 |       pragma Loop_Invariant (Prop);
      |                              ^~~~
```

Figure 5: Loop Invariants–Failure at arbitrary iteration [4]

Loop Variants are a somewhat simpler version of Loop Invariants [4]. Loop Variants check that a scalar increases or decreases for each iteration of the loop [51]. This is usually used to communicate to the prover, GNATprove, that a variable is bounded and does not increase or decrease an infinite number of times[5]. Loop Variants are a very effective tool when trying to prove the termination of a loop, as shown in Figure 6.

```
procedure Terminating_Loops (X : Natural) with
  SPARK_Mode
is
   Y : Natural;
begin
   Y := 0;
   while X - Y >= 3 loop
      Y := Y + 3;
      pragma Loop_Variant (Increases => Y);
   end loop;

   Y := 0;
   while X - Y >= 3 loop
      Y := Y + 3;
      pragma Loop_Variant (Decreases => X - Y);
   end loop;
end Terminating_Loops;
```

Figure 6: Loop Variants [4]

III. Verification of Resampling Methods

3.1 Preamble

The primary goals of this work are as follows:

- Explain the relationships that were discovered between the sampling array and the particle weights array in systematic and stratified resampling
- Prove the discovered relationships in systematic and stratified resampling through SPARK
- Explain the relationships created in multinomial resampling, residual resampling, and why the proofs created are out of the scope of this paper
- Walkthrough the code of the simulation that will be used in Chapter IV

This chapter provides a detailed description of the methodology used to prove the resampling methods discussed in this paper. It begins by describing the relationships created between the systematic sampling array and the particle weights array. Following that, we prove that these relationships are true and hold for an arbitrary number of iterations with the use of SPARK. The same two steps above are then used to prove all the relationships that we discovered in stratified resampling. The chapter will also discuss the difference between implicit and explicit relationships, along with why the logic behind multinomial and residual resampling create relationships that are not worth proving. The chapter will conclude with walking through the code for the simulation that will be discussed in Chapter IV.

3.2 Construction of Systematic Resampling Proof

```
pragma SPARK_Mode;
type F_Arr is array (0 .. 499) of Float;
type I_Arr is array (0 .. 499) of Integer;
big_int : constant Integer := 2000000;
max_particles : constant Integer := 500;
subtype Positive is Integer range 0 .. big_int*max_particles;
subtype rand_range is Integer range 1 .. big_int;
```

Variables in helper_method File

The verification of the systematic resampling algorithm is split between two different files: resample and helper_methods. The resample file includes all of the logic and proofs for the resampling algorithm, while the helper_methods file is used to create the floating point error corrected CDF array of the weights and sampling array. In order to fully understand the algorithm, it is best to start with the proofs and logic of the helping_methods file. Some of the variables will need to be explained later in order to grasp their true purpose. The most important ones right now are the two array type variables: F_Arr and I_Arr. F_Arr is a float array of length 500. F_Arr will be used for representing the weight of each particle; since the length is 500, there will be 500 particles. Having a set number of particles does not affect the credibility of this proof as changing the number of particles has no effect on the particle filters logic. I_Arr is an integer array of length 500 and it has no specific implementation during the algorithm.

The first hurdle of the proof was dealing with the floating point error from the array of weights that is passed into the resampling method. While the floating point error in the weights array is not significant as a PDF, it becomes an issue when the weights array is converted into a CDF because the error will accumulate and present a possibility of having a final value that is not 1.0. As a reminder, mathematically CDFs always have a final value of exactly 1.0, because it is the sum of all of the past probabilities. Most implementations of a resampling method will counteract this error by hard coding the last index to be equal to 1.0. This is an insufficient solution as

it does not account for the cases when the CDF becomes 1.0 before its final value. Mathematically, it is a rule for a CDF to have a final value of one but for the actual implementation of our resampling method it does not have a major effect on the output. The reason for this is because we have a relatively small number of particles, so when making the CDF the floating point error does not become large enough to cause any real concern. Nevertheless, when dealing with proofs the mathematical integrity of an algorithm does matter, as we are trying to prove that an algorithm always does what it is supposed to do. In order to create a definitive range for a CDF array it is in our best interest to move our weights to the integer domain as fixed points.

```
--ADB File:
function int_weights
(A : F_Arr)
return I_Arr
is
int_weights_arr : I_Arr := (A'First..A'Last => 0);
begin
for I in A'Range loop
int_weights_arr (I) := Integer(Float(A'Length * big_int) * A(I));
if int_weights_arr (I) > big_int * A'Length then
int_weights_arr (I) := big_int * A'Length;
elsif int_weights_arr (I) < 0 then
int_weights_arr (I) := 0;
end if;
pragma Loop_Invariant (for all K in int_weights_arr'First .. I =>
int_weights_arr (K) in 0 .. big_int * A'Length);
end loop;
return int_weights_arr;
end int_weights;

--ADS File:
function int_weights(A : F_Arr) return I_Arr with
Pre => (for all I in A'Range => A(I) in 0.0 .. 1.0),
Post => (for all J in int_weights'Result'Range => int_weights'Result (J) <= big_int*A'Length and
    then
int_weights'Result (J) >= 0);
```

int_weights Function

The purpose of the int_weights function can be shown through the pre and post conditions in Appendix A. The preconditions make sure that a PDF is being past

into the functions, while the post condition bound the range of the returning I_Arr between 0 and Big_int*N (N being the number of particles/length of the arrays). The returning array of the int_weights function is an integer array representation of the passed in float array. As shown in Appendix A, this is done by multiplying every element in the weights array by Big_int*N and then converting every value into an integer. If any of the elements end up being less than 0 or greater than Big_int*N, the int_weights function will hard code the element to 0 or Big_int*N respectively. It is important to notice that our integer representation of 1.0 is now Big_int*N because it is only possible to get this value when the float point value of a weight that was passed in is equal to 1.0 (A particle with a weight of 1.0 means we are 100% confident in said particle). Eq (2) shows the equation that is used to generate the output of int_weights. At timestep=0, the weights array is equal to 1/N because we do not have any confidence in any specific particle so they are all weighted evenly. Over time, the weights array changes because it is multiplied by and then normalized. In our int_weights function, we take the weights array and multiply it by Big_int*N, which gives us the array shown in Eq (2).

$$weights_0 = \frac{1}{N}$$

$$weights_n = \frac{\frac{1}{N} * P(z|x)}{\sum_{I=0}^{499} weights[I]}$$

$$int_weights = Big_int * N * weights_n$$

$$int_weights = Integer(\frac{Big_int * P(z|x)}{\sum_{I=0}^{499} weights[I]}) \tag{2}$$

Int_weights is a simple solution to converting all of the weights over to the integer domain but it does not solve the problems caused by the floating point error. In addition, we have now introduced the issue of cutting off part of the floats when

converting them to integers. These errors mean that when we create the CDF of this int_weights array our final element is not guaranteed to be Big_int*N (Our integer representations of 1.0). In order to make sure a mathematically correct version of int_weights is used inside of the systematic resampling method additional steps must be taken.

```
--ADB File:
function CDF
(A : I_Arr)
return I_Arr
is
Result : I_Arr := (A'First..A'Last => 0);
begin

Result (A'First) := A(A'First);
for I in A'First + 1 .. A'Last loop
Result (I) := Result (I-1) + A (I);
if Result (I) > big_int * A'Length then
Result (I) := big_int * A'Length;
elsif Result (I) < 0 then
Result (I) := 0;
end if;
pragma Loop_Invariant(Result (A'First) = A(A'First));

pragma Loop_Invariant(for all K in Result'Range => Result (K) >= 0 and
Result (K) <= big_int * A'Length);

pragma Loop_Invariant(for all K in Result'First .. I =>
(for all J in Result'First .. K => Result (K) >= Result (J)));
end loop;
Result(Result'Last) := big_int * A'Length;
return Result;
end CDF;

function PDF
(A : I_Arr)
return I_Arr
is
CDF_A : I_Arr := CDF(A);
Result : I_Arr := (A'First..A'Last => 0);
begin

Result (A'First) := A(A'First);
for I in Result'Range loop
if I = 0 then
Result(I) := CDF_A(I);
else
Result(I) := CDF_A(I) - CDF_A(I-1);
end if;
pragma Loop_Invariant(Result (A'First) = A(A'First));
```

```
45
46 pragma Loop_Invariant(for all K in Result'Range => Result (K) >= 0 and
47 Result (K) <= big_int * A'Length);
48
49 pragma Loop_Invariant(for all K in Result'First .. I =>
50 (if K = 0 then Result(K) = CDF_A(K) else Result(K) = CDF_A(K) - CDF_A(K-1)));
51 end loop;
52 return Result;
53 end PDF;
54
55 --ADS File:
56 function CDF (A : I_Arr) return I_Arr with
57 Pre => (for all J in A'Range => A(J) <= big_int*A'Length and A(J) >= 0),
58 Post => A (A'First) = CDF'Result(CDF'Result'First) and then
59 CDF'Result(CDF'Result'Last) = big_int*A'Length and then
60 (for all I in CDF'Result'Range => CDF'Result (I) >= 0 and then
61 CDF'Result (I) <= big_int*A'Length) and then
62 (for all K in CDF'Result'Range => (for all J in CDF'Result'First .. K => CDF'Result (K) >= CDF'
     Result (J)));
63
64 function PDF (A : I_Arr) return I_Arr with
65 Pre => (for all J in A'Range => A(J) <= big_int*A'Length and A(J) >= 0),
66 Post => A (A'First) = CDF(A)(CDF(A)'First) and then
67 (for all I in PDF'Result'Range => PDF'Result (I) >= 0 and then
68 PDF'Result (I) <= big_int*A'Length) and then
69 (for all K in PDF'Result'Range => (if K = 0 then PDF'Result(K) = CDF(A)(K)
70 else PDF'Result(K) = CDF(A)(K) - CDF(A)(K-1)));
```

CDF and PDF Function

The next function in the helper methods file is the CDF function which, given a I_Arr, will create a CDF array with three important post conditions shown in Appendix A. The first post condition makes sure that the first index is equal to the PDF that was passed in, the second post condition makes sure that each index is between 0 and N*Big_int, and the last post condition makes sure that each element is greater than or equal to all of the previous elements. Similar to the int_weights function, this function also makes sure that if any element is greater than N*Big_int or less than 0 it will hard code it to N*Big_int or 0 respectively. Similar to the solution in the floating point domain, the last element is hard coded to always be equal to N*Big_int. This has not fixed our issues that came out of the int_weights function but it does provide us with the tools we need to create a mathematically

correct int_weights array. The way we create a floating point error free int_weights array is by taking the CDF that we just created and creating a PDF out of it. If the PDF is created from the CDF then the rule that the CDF is the cumulative sum of the PDF holds. This PDF will be the one that is passed into the systematic resampling algorithm. The main problem with the PDF is that it has been changed from the original floating point version that was passed into the function. To review, the weights were converted into integers which cut off part of each float, the bounds and final element of the CDF were hard coded; all of these create a difference between the weights array and the PDF passed into the resampling method. Thankfully, these differences do not cause any problems in the code for the same reason that the floating point error did not. The traditional way of setting the final element equal to 1.0 in the floating point domain mentioned above, did not cause anything of significance to happen because the whole point of the weights is to make more copies of weights that are large and less copies of weights that are small. The same logic that was used in the floating point domain is now used in our integer domain representation but now we do not have the issue of there being anything outside of the range of 0 and N*Big_int. All of this leads to the next function in the helper_methods file: PDF function which, as mentioned above, creates a PDF out of the CDF. The code for this function is shown above and the post conditions are the same as the int_weights function, aside from checking that the relationship between the PDF and the CDF holds. Unless stated otherwise, any mention of weights, CDF, or PDF are directed towards the floating point error corrected version of them.

```
1 --ADB File:
2 function sys_samp_arr
3 (offset : Integer)
4 return I_Arr
5 is
6 steps : I_Arr;
7 begin
8 steps := (steps'First..steps'Last => 0);
9 for I in steps'First .. steps'Last loop
```

```
steps (I) := I*big_int + offset;
pragma Loop_Invariant(for all J in steps'First .. I => steps (J) = J*big_int + offset);

pragma Loop_Invariant(for all J in steps'First+1 .. I =>
(for all K in steps'First..J-1 => steps(J) > steps(K)));
end loop;
return steps;
end sys_samp_arr;


--ADS File:
function sys_samp_arr (offset : Integer) return I_Arr with
Pre => offset >= 1 and then offset <= big_int,
Post => sys_samp_arr'Result (sys_samp_arr'Result'First) = offset and then
(for all J in sys_samp_arr'Result'Range => sys_samp_arr'Result (J) = J*big_int + offset) and then
sys_samp_arr'Result (sys_samp_arr'Result'Last) <= sys_samp_arr'Result'Length*big_int and then
(for all J in sys_samp_arr'Result'First+1 .. sys_samp_arr'Result'Last =>
(for all K in sys_samp_arr'Result'First..J-1 => sys_samp_arr'Result(J) > sys_samp_arr'Result(K)));
```

sys_samp_arr Function

The last helper method is the sys_samp_arr, which creates the systematic sampling array that will be used to sample from the CDF of the weights. In the float point domain, the sampling array follows Eq (3)

$$\sum_{I=0}^{499} \frac{I}{N} + Random_offset \tag{3}$$

$$\sum_{I=0}^{499} Big_int * I + Big_int * Random_offset \tag{4}$$

and when we convert it over to the integer domain representation we always multiply by N*Big_int which gives us Eq (4). The sys_samp_arr function creates a sampling array following Eq (4) shown above. First, we check that each element is equal to I*big_int + offset. Second, we state that the upper limit of the sampling array is N*big_int. This is because the highest value of I is 499 and the highest value of the offset, as stated in the precondition, is Big_int. Lastly, we make sure to state that every value in the sampling array is equal to or greater than the previous values.

Now that the helper_methods file has been explained, it is time to move on to the

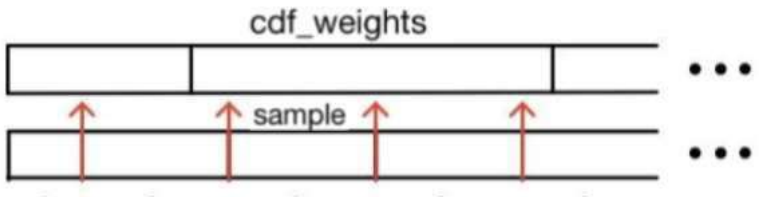


Figure 7: Systematic Sampling Array

resample file which holds the logic and proofs for the systematic resampling algorithm. Figure 7 shows a visual representation of the arrays we made in the helper_methods file. It is important to understand that the boxes represent the difference between each element and not the contents. The contents of the array can be seen below each line; each line is where a sample will be taken. In other words, every line in the sampling array is a sample that will be taken from the cdf_weights array.

```
--ADB File:
function systematic_resampling (random_number : Integer;
I_Weights : I_Arr)
return I_Arr
is
dup_arr : I_Arr := (others => 0);
cdf_weights : I_Arr := CDF(I_Weights);
sample : I_Arr := sys_samp_arr(random_number);
begin

for I in sample'Range loop
if I > sample'First then
dup_arr (I) := dup_arr (I-1);
end if;

while sample(I) > cdf_weights(dup_arr(I)) loop
dup_arr (I) := dup_arr (I) + 1;
end loop;

end loop;
return dup_arr;
end systematic_resampling;
```

systematic_resampling Function

In the code, we start off with making the sampling array and the CDF of the weights from which we are going to sample. The algorithm itself is quite simple but

proving that it works the way it is supposed to be is far from intuitive. The first for-loop iterates through the sample array (sample). Inside the for-loop a while loop is used to iterate through cdf_weights until it finds an index where it is greater than sample[I], that index is stored inside of dup_arr[I] which is also the array we are returning. The return array will be used as a mask to create the new particles array as shown below. The returning array, dup_arr, will be represented by the variable φ and element dup_arr[I] will be represented by φ_I. Any arbitrary element, φ_I, follows the rule that cdf_weights[φ_I]¿=sample[I]. The significance of storing an index inside of dup_arr[I] is that we are sampling it and stating that this particle is statistically significant enough to keep around.

dup_arr = [3, 4 , 4 , 3, 3]
particles = [p1 ,p2 ,p3 ,p4 ,p5]
new_particles = [p4, p5, p5, p4, p4]

When analyzing the sampling array Eq (4), you can make the observation that if the weight of a particle is greater than or equal to Big_int it has to be sampled at least once. While this statement is necessary in describing how this algorithm works it is not sufficient. If we were to take it a step further and say that the size of a specified weight is greater than or equal to 2*big_int but less than 3*big_int then we can say that the minimum number of samples that can be taken from this particle is equal to 2. This formula holds for any size weight, even in the maximum case of when the weight is equal to 500*big_int because this means that 500 samples will be taken from this weight. Logically, this makes perfect sense because the maximum value in the sample array is 500*big_int and if one weight is equal to 500*big_int then that means the rest of the weights are equal to 0 (500*big_int is our fixed-point

representation of 1.0). This same logic can be used to find a maximum value for the number of samples that can be taken from a weight. Again, if the weight of a particle is between 2*big_int but less than 3*big_int it is not possible for it to be sampled more than 3 times.

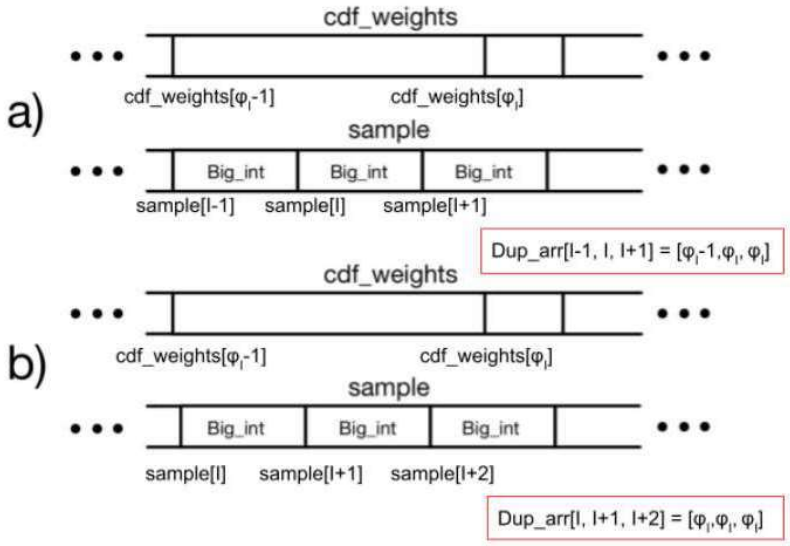


Figure 8: Minimum and Maximum for Systematic Sampling

Figure 8 shows an example of the minimum and maximum number of samples possible for a given weight. Mathematically, each block in the sample bar is equal to sample[I] - sample[I-1] and each block in the cdf_weights bar is equal to cdf_weights[I]-cdf_weights[I-1], which is just the PDF. For the minimum edge case to happen in Figure 4a, sample[I] has to be at its maximum value for sampling cdf_weights[φ_I]. The rule when we use the index φ_I is that sample[I] less than or equal to cdf_weights so the maximum value of a sample taken from element cdf_weights[φ_I] is cdf_weights[φ_I]. When we look at the case where we are looking at the very first sample taken of an element the maximum value changes. The smallest value sample that can be taken from cdf_weights[φ_I] is cdf_weights[φ_I-1] + 1 because sample[I] is greater than cdf_weights[φ_I-1] but less than or equal to cdf_weights[φ_I]. The largest value sample

that can be taken from cdf_weights[φ_I] is cdf_weights[φ_I-1] + Big_int because anything greater would mean that sample[I-1] (which always has a value of sample[I]-Big_int) would also be sampling cdf_weights[φ_I], and we stated that this is the first sample taken. In figure 8a, we see an example of the largest value sample possible. Notice the value of sample[I-1] in this case, it is equal to cdf_weights[φ_I-1]. Sample[I-1] will always be equal to cdf_weights[φ_I-1] when looking at the maximum first sample case (when sample[I] equals cdf_weights[φ_I-1]+Big_int) because, again, sample[I-1] equals sample[I]-Big_int.

In figure 8b, we look at the smallest value sample that can be taken from cdf_weights[φ_I] which, as mentioned above, is where sample[I] equals cdf_weights[φ_I-1] + 1. In this case, the only rule that can be said about the relationship between cdf_weights[φ_I-1] and sample[I-1] is that sample[I-1] equals cdf_weights[φ_I-1] + 1-Big_int. These relationships between the sampling and cdf_weight array are crucial in proving the systematic resampling method.

In order to prove that the logic described above stays consistent for all cases in systematic resampling a mathematical formula for determining the range of the number of duplicates that could be made for any specified weight is necessary:

$$Min : \#ofsamples = \lfloor \frac{weights[\varphi_I]}{Big_int} \rfloor \tag{5}$$

$$Max : \#ofsamples = \lfloor \frac{weights[\varphi_I]}{Big_int} \rfloor + 1 \tag{6}$$

The right side of Eq (5) and (6) is an integer divide of the weight by Big_int. As described in the explanation of figure 8, this formula means that the minimum number of samples taken of a weight is the number of times Big_int can fit into it. The maximum number of samples taken of a weight is the number of times Big_int can

fit into it plus one because of the random offset in the first elements of the sampling array.

3.3 Formal SPARK Verification of Systematic Resampling

```
1  --ADB File:
2  function systematic_resampling (random_number : Integer;
3  I_Weights : I_Arr)
4  return I_Arr
5  is
6  dup_arr : I_Arr := (others => 0);
7  cdf_weights : I_Arr := CDF(I_Weights);
8  sample : I_Arr := sys_samp_arr(random_number);
9  begin
10
11 for I in sample'Range loop
12 if I > sample'First then
13 dup_arr (I) := dup_arr (I-1);
14 end if;
15
16 while sample(I) > cdf_weights(dup_arr(I)) loop
17 dup_arr (I) := dup_arr (I) + 1;
18 pragma Loop_Variant(Increases => dup_arr(I)); --(I.1)
19
20 pragma Loop_Invariant(sample(sample'Last) <= cdf_weights(cdf_weights'Last));--(I.2)
21
22 pragma Loop_Invariant(dup_arr(I) in cdf_weights'Range);--(I.3)
23
24 pragma Loop_Invariant(if I = sample'First then dup_arr(I) > dup_arr'Loop_Entry(I) else dup_arr(I)
       > dup_arr(I-1));--(I.4)
25
26 pragma Loop_Invariant(sample(I) > cdf_weights(dup_arr(I)-1)
27 and then sample(I) <= cdf_weights(dup_arr(I)-1) + big_int);--(I.5)
28 end loop;
29
30 pragma Loop_Invariant(for all K in dup_arr'Range => dup_arr(K) in cdf_weights'Range);--(O.1)
31
32 pragma Loop_Invariant(for all K in dup_arr'First+1..I => (for all J in dup_arr'First..K-1 =>
       dup_arr(K) >= dup_arr(J)));--(O.2)
33
34 pragma Loop_Invariant(if dup_arr(I) = 0 then sample(I) <= cdf_weights(dup_arr(I)) else
35 sample(I) > cdf_weights(dup_arr(I)-1) and then sample(I) <= cdf_weights(dup_arr(I)));--(O.3)
36
37 pragma Loop_Invariant(for all K in dup_arr'First..I =>
38 (if dup_arr(K) > 0 and then K = 0 then sample(K) - cdf_weights(dup_arr(K)-1) in 1 .. big_int)
39 and then (if dup_arr(K) > 0 and then K > 0 and then dup_arr(K-1) /= dup_arr(K)
40 then sample(K) - cdf_weights(dup_arr(K)-1) in 1 .. big_int));--(O.4)
41
42 pragma Loop_Invariant(for all K in sample'First..I =>
43 (for all J in sample'First..K =>
44 (if dup_arr(K) > 0 and then dup_arr(J) = dup_arr(K) then
```

```
sample(J) <= cdf_weights(dup_arr(K)) and then
sample(J) > cdf_weights(dup_arr(K)-1)) and then
(if dup_arr(K) = 0 and then dup_arr(J) = dup_arr(K) then
sample(J) <= cdf_weights(dup_arr(K)))));--(O.5)

pragma Loop_Invariant(for all K in dup_arr'First+1..I =>
(for all J in dup_arr'First..K =>
(if dup_arr(K-1) = dup_arr(J) and then dup_arr(K) /= dup_arr(J) then
(if J = 0 or (J > 0 and then dup_arr(J) > dup_arr(J-1)) then
K-J in I_Weights(dup_arr(J))/big_int ..
I_Weights(dup_arr(J))/big_int + 1)) and then

(if K = sample'Last and then dup_arr(K) = dup_arr(J) then
(if J = 0 or (J > 0 and then dup_arr(J) > dup_arr(J-1)) then
K+1-J in I_Weights(dup_arr(J))/big_int ..
I_Weights(dup_arr(J))/big_int + 1))));--(O.6)


end loop;



return dup_arr;
end systematic_resampling;


--ADS File:
function systematic_resampling (random_number : Integer; I_Weights : I_Arr) return I_Arr with
Pre => I_Weights'First = 0 and then I_Weights'Last+1 <= max_particles and then
random_number >= 1 and then random_number <= big_int and then
(for all J in I_Weights'Range => I_Weights(J) in 0 .. big_int*I_Weights'Length) and then
I_Weights (I_Weights'First) = CDF(I_Weights)(I_Weights'First) and then
(for all J in I_Weights'Range => (if J = 0 then I_Weights(J) = CDF(I_Weights)(J)
else I_Weights(J) = CDF(I_Weights)(J) - CDF(I_Weights)(J-1))),

Post => (for all K in systematic_resampling'Result'First+1..systematic_resampling'Result'Last =>
(for all J in systematic_resampling'Result'First..K =>

(if systematic_resampling'Result(K-1) = systematic_resampling'Result(J) and then
systematic_resampling'Result(K) /= systematic_resampling'Result(J) then
(if J = 0 or (J > 0 and then systematic_resampling'Result(J) > systematic_resampling'Result(J-1))
    then
K-J in I_Weights(systematic_resampling'Result(J))/big_int ..
I_Weights(systematic_resampling'Result(J))/big_int + 1)) and then

(if K = systematic_resampling'Result'Last and then
systematic_resampling'Result(K) = systematic_resampling'Result(J) then
(if J = 0 or (J > 0 and then systematic_resampling'Result(J) > systematic_resampling'Result(J-1))
    then
K+1-J in I_Weights(systematic_resampling'Result(J))/big_int ..
I_Weights(systematic_resampling'Result(J))/big_int + 1))));
```

systematic_resampling Function

In order to give spark the information it needs to verify Eq (5) and (6), there are several loop invariants required. The nature of the outer for-loop in the code above is to iterate through samples with iterator I. The iterator, I, has a range between 0 and 499 because there are 500 particles. The inner while loop is responsible for iterating through cdf_weights until it finds an index where cdf_weights[φ_I] is greater than sample[I]. Both loops require descriptive loop invariants in order to prove our post condition that will be created from Eq (5) and (6). The only other piece of code in the algorithm that is not for the proof is setting the value of dup_arr[I] equal to dup_arr[I-1]; the reason for this is because we know that the next sample has to be at the same or a high index in the cdf_weights array.

The inner while loops first loop variant, (I.1), states that dup_arr[I] increases which is true because we are iteratively searching for the index while in this loop so it will always be increasing while inside of it. (I.2) is used to tell the compiler that the while loop will eventually exit, it does this by stating that the last element of the sample array is less than or equal to the last element of the cdf_weights array. The max value of Eq (6) is 499*Big_int + Big_int which is the last element of our cdf_weights array (500*Big_int is our integer representation of 1.0). The condition for being in the while loop is that sample[I] is greater than cdf_weights[φ_I] which will eventually break because of the last loop invariant. (I.3) makes sure that cdf_weights[φ_I] never gets an out of bound index. Since the most recent loop invariant guaranteed that we would exit the while loop we know that φ_I will never go out of bounds. Similar to how sample and cdf_weights elements get greater in value as we get farther into each of the arrays, φ has that very same property. All elements in φ will either be greater than or equal to the past values, this is the reason for (I.4) which states that all values in φ are greater than or equal to the past values. Next we establish the relationship between sample[I] and cdf_weights[φ_I-1] with the last inner loop invariant, (I.5). As

mentioned in the previous section, when the first sample of a weight is taken the value of sample[I] can be characterized in terms of cdf_weights[φ_I-1] to form a range: Sample[I] can be between cdf_weights[φ_I-1] + 1 and cdf_weights[φ_I-1] + Big_int. The reason that this loop invariant works inside of the inner whole loop but not the outer for loop is because if we are entering the inner while loop φ_I does not equal φ_I-1 (φ_I-1 and φ_{I-1} are different because φ_I-1 is the element before φ_I and φ_{I-1} is the element before φ_I that is inside of φ; φ_I could be equal to or greater than φ_I-1). If φ_I does not equal φ_I-1 then that means that it is the first sample being taken of cdf_weights[φ_I], when we are outside of the inner loop this is not guaranteed.

The purpose of the outer loop is to iterate through the sample array until we have taken as many samples as there are elements but now we need to prove that Eq (5) and (6) hold for each iteration. The (O.1) and (O.2) are there to make sure that (I.2) and (I.3) hold when in the outer loop. Even though there is no code that changes (I.2) and (I.3) in the outer loop, the compiler does not know this because it is in a for loop and we need a loop invariant to tell it what is going on. (O.3) relates directly to (I.5) but the upper bound has changed to include the case when φ_I equals φ_I-1; this means that it now includes the case where it is not the first sample taken of cdf_weights[φ_I]. When including the case of it not being the first sample of a weight, all that can be said about the upper bound of sample[I] is that it is less than or equal to cdf_weights[φ_I]. (O.3) is always true because it comes after the inner while loop. The compiler needs to still understand the relationship from (I.5) remains unchanged when iterating through the outer loop but, as we said in (O.3), (I.5) only occurs when it is the first sample being taken of a weight. (O.4) uses multiple if statements to describe (I.5)'s meaning outside of the inner while loop. When we are looking at the first iteration of the while loop all that we have to check is that φ_I is greater than 0 because we are going to be using cdf_weights[φ_I-1]. When

it is the first iteration of the outer loop and φ_I is greater than 0, it means that it is the first sample being taken and that sample[0] is greater than cdf_weights[φ_0]. The only case when (I.5) holds in the outer loop when it is not the first iteration is when it is the first sample being taken of a weight. The conditional statements we use in (O.4) to check the case mentioned above is when the iterator is greater than 0, φ_I is greater than 0, and φ_I is greater than φ_I-1. (O.5) is a very important loop invariant because it characterizes the most unique trait of the φ array: when an arbitrary element φ_J equals another arbitrary element φ_K, every element in between the two is equal. (O.5) handles the case of when multiple samples are taken of the same weight. The outer for loop of the loop invariant, K, can range between 0 and I. The purpose of K is to act as an upper bound, where if the lower bound φ_J (the inner loop of the loop invariant) is equal to φ_K, everything in between is also equal. That does not mean that J and K are equal, for example, lets say that J is the second sample being taken in a sampling array and K is the 4 sample being taken but both are sampling the same element in the cdf_weights array (the elements J and K are sampling is represented by φ_J and φ_K). The if statements inside of the for loops is similar to the logic of (O.4) because it handles when we are and are not on the first element of φ. When it is not the first element and φ_J equals φ_K there are two statements that can be said about the relationship between the sample and cdf_weights array: sample[J] is less than or equal to cdf_weights[φ_K] and sample[J] is greater than cdf_weights[φ_K-1]. The first statement is correct because φ_J equals φ_K and the inner while loop before the loop invariants makes sure that sample[J] is less than or equal to cdf_weights[φ_K]. The second statement is correct because we know that cdf_weights[φ_J] equals cdf_weights[φ_K] and sample[J] is greater than cdf_weights[φ_J-1], so when we interchange the two formulas we get that sample[J] is greater than cdf_weights[φ_K-1]. When it is the first element we can no longer use the

index cdf_weights[φ_K-1] because that is out of bounds but we can say that sample[J] is less than or equal to cdf_weights[φ_K] for the same reasoning mentioned above.

The final loop invariant, (O.6), is what will be proving Eq (5) and (6). The loop invariants prior to (O.6) are all required in order for the compiler to understand it. In order to put Eq (5) and (6) in terms of code, two for loops are require. Similar to (O.5) the outer for loop, K, will serve as the upper bound and the inner for loop, J, will serve as the lower bound. We want to design the for loops so that the value of K-J is equal to the number of copies that are made of cdf_weights[φ_J]. In order to get K-J to equal the number of copies made of cdf_weights[φ_J] we need the first two if statements to be true: φ_J equals φ_K-1 and φ_K does not equal φ_J. This will make it so K-J equals the number of copies. Now that we have the number of copies we need to use it in Eq (5) and (6) which can be coded with the use of the PDF (the I_weights that were passed into the systematic resampling function). The final result is that K-J is in between the range of I_weights[φ_J]/Big_int and I_weights[φ_J]/Big_int + 1 which lines up directly to Eq (5) and (6). The next part of the for loop is proving the exact same rule but it is for when we are at the end of the loop and φ_J equals φ_K. This is an edge case situation but it needs to be address as it changes the upper bound, in order to counter act this change we need to add 1 to K-J (the number of copies of element φ_J is now equal to K-J+1). The post condition for systematic resampling is a rewritten version of (O.6) as that is what we wanted to prove.

3.4 Construction of Stratified Resampling Proof

The verification of the stratified resampling algorithm is in the same format as systematic resampling, it is split between two different files: resample and helper_methods. The resample file includes all of the logic and proofs for the resampling algorithm, while the helper_methods file is used to create the floating point error corrected CDF

array of the weights and sampling array. In order to fully understand the algorithm, it is best to start with the proofs and logic of the helping_methods file. The helper_methods file is the same as the helper_methods file in systematic resampling, all of the same methods will be used in addition to a new method called strat_samp_arr. Strat_samp_arr creates the stratified sampling array that will be used to sample from the CDF of the weights. In the float point domain, the sampling array follows Eq (7)

$$\sum_{I=0}^{499} \frac{I}{N} + Random_offset[I] \tag{7}$$

$$\sum_{I=0}^{499} Big_int * I + Big_int * Random_offset[I] \tag{8}$$

```
--ADB File:
function strat_samp_arr
(offset_arr : I_Arr)
return I_Arr
is
steps : I_Arr;
begin
steps := (steps'First..steps'Last => 0);
for I in steps'First .. steps'Last loop
steps (I) := I*big_int + offset_arr (I);
pragma Loop_Invariant(for all J in steps'First .. I => steps (J) = J*big_int + offset_arr (J));

pragma Loop_Invariant(for all K in steps'First .. I =>
(if K = steps'First then steps(K) in 1 .. big_int
else steps(K) in K*big_int + 1 .. K*big_int + big_int));

pragma Loop_Invariant(for all J in steps'First+1 .. I =>
(for all K in steps'First .. J-1 =>
steps (J) - steps(K) in (J-K-1)*big_int+1 .. (J-K+1)*(big_int)-1));

pragma Loop_Invariant(for all K in steps'First+1 .. I =>
(if offset_arr(K-1) = 1 and then offset_arr(K) = big_int
then steps(K) - steps(K-1) = 2*big_int-1));
end loop;
return steps;
end strat_samp_arr;

--ADS File:
function strat_samp_arr (offset_arr : I_Arr) return I_Arr with
Pre => (for all K in offset_arr'Range => offset_arr (K) in 1 .. big_int),
```

```
31 Post => strat_samp_arr'Result (strat_samp_arr'Result'First) = offset_arr(offset_arr'first) and
       then
32 strat_samp_arr'Result (strat_samp_arr'Result'Last) <= strat_samp_arr'Result'Length*big_int and
       then
33 (for all J in strat_samp_arr'Result'Range => strat_samp_arr'Result (J) = J*big_int + offset_arr(J)
       ) and then
34 (for all K in strat_samp_arr'Result'Range =>
35 (if K = strat_samp_arr'Result'First then strat_samp_arr'Result(K) in 1 .. big_int
36 else strat_samp_arr'Result(K) in K*big_int + 1 .. K*big_int + big_int)) and then
37 (for all J in strat_samp_arr'Result'First+1 .. strat_samp_arr'Result'Last =>
38 (for all K in strat_samp_arr'Result'First .. J-1 =>
39 strat_samp_arr'Result (J) - strat_samp_arr'Result (K) in (J-K-1)*big_int+1 .. (J-K+1)*(big_int)-1)
       );
```

strat_samp_arr Function

When we convert it over to the integer domain representation we always multiply by N*Big_int which gives us Eq (8). The strat_samp_arr function creates a sampling array following Eq (8) shown in the code above. First, we check that each element is equal to I*big_int + offset. Second, we state that the upper limit of the sampling array is N*big_int. This is because the highest value of I is 499 and the highest value of the offset, as stated in the precondition, is Big_int. Lastly, we make sure to state that every value in the sampling array is equal to or greater than the previous values. In chapter 2, we mentioned that the range for our Random_offset array was $(0, \frac{1}{N}]$. After the conversion to the integer domain, where we multiply all floats by Big_int * N, the new range for the Random_offset array is [1, Big_int]. Given the range for the Random offset array we can make a range for every index of the sampling array: [Big_int * I + 1, Big_int * I + Big_int]. This is implicitly checked in the post condition with checking that the formula $\sum_{I=0}^{499} Big_int * I + Big_int * Random_offset[I]$ holds, the only difference is that we are checking the exact value of each index in Random offset array.

Now that the helper_methods file for Stratified Resampling has been explained, it is time to move on to the resample file which holds the logic and proofs for the stratified resampling algorithm. Majority of this file is the exact same as the sys-



Figure 9: Stratified Sampling Array

tematic resampling verification, all of the loop invariants are the same aside from some minor changes in each value. Nonetheless, these proofs were significantly harder to prove because of the difficulty in understanding why they are similar and how the post condition will change when a different random value is added to every index instead of the same value. The best way to understand stratification is with a visual (Figure 9) that is similar to the visual used in systematic resampling. Figure 9 shows the range of possible samples per index that was discussed in the stratified resampling section in chapter 2. The interesting part of stratified resampling discussed in chapter 2 was that the range of the current index affects the range of the next index. That is, when we are not given any other information about a index we can say that the interval between two samples is $(sampling_array[I] - sampling_array[I-1]) \in (FixedInterval, 2 * FixedInterval) = [1, Big_int * 2]$. The reason it has changes from (0, fixed interval * 2) is because we have converted over to the integer domain and the float point domain fixed interval was 1/N. When we are given information about the previous index we can make a new range based on that information. If the previous sample was sampled at its max value (sampling_array[I-1] = I * Big_int + Big_int) then the range for the next index would be $sampling_array[I+1] - sampling_array[I] \in [1, Big_int]$ because we know $sampling_array[I+1] \in [(I+1) * Big_int + 1, (I+1) * Big_int + Big_int]$ and we also know that sampling_array[I] = I * Big_int + Big_int, which creates the range [(I+1) * Big_int + 1 - (I * Big_int + Big_int), (I+1) * Big_int + Big_int - (I *

Big_int + Big_int)] = [1, Big_int]. If the previous sample was sampled at its min value (sampling_array[I-1] = I * Big_int + 1) then the range for the next index would be $sampling_array[I+1] - sampling_array[I] \in [Big_int, Big_int * 2]$ because we know $sampling_array[I+1] \in [(I+1) * Big_int + 1, (I+1) * Big_int + Big_int]$ and we also know that sampling_array[I] = I * Big_int + 1, which creates the range [(I+1) * Big_int + 1 - (I * Big_int + 1), (I+1) * Big_int + Big_int - (I * Big_int + 1)] = [Big_int, Big_int * 2]. The reason for creating a range for when we given information about the previous samples is because, while saying that the range between each sample is in [1, Big_int * 2] is necessary, it is not sufficient in describing all the aspects of this sampling method. The proof that we create for stratified resampling will need to be of the same format as systematic resampling but the range will need to be altered to fit the characteristic described above.

In systematic resampling we proved that every weight will be sampled between [I_weights[φ_J]/Big_int, I_weights[φ_J]/Big_int + 1]. In stratified resampling we can assume that the range for the number of copies will increase because the interval between each sample has changed from the constant Big_int (systematic resampling has a set interval of sampling every Big_int) to a range between [1, Big_int * 2]. The range gives us information about how the proof ([I_weights[φ_J]/Big_int, I_weights[φ_J]/Big_int + 1]) will change for stratified resampling: the possibility of sampling_array[I] - sampling_array[I-1] = 1 means that it is possible for a weight to be sampled twice when its weight is less than Big_int and the the possibility of sampling_array[I] - sampling_array[I-1] = Big_int * 2 means that it is possible for a weight to be sampled zeros times when its size is less than Big_int * 2. For creating the new proof for stratified resampling we have to also consider what happens after the min or max case happens. Let's consider a case where the size of the weight equals Big_int * 4 + 2, what would the minimum and maximum number of samples taken be? For

the maximum case, we would set sampling_array[I] - sampling_array[I-1] = 1 but this means that the next minimum interval is sampling_array[I+1] - sampling_array[I] = Big_int. As discussed before, the reason that sampling_array[I+1] - sampling_array[I] = Big_int is because we know that sampling_array[I] = I * Big_int + 1 and that sampling_array[I-1] = (I-1) * Big_int + Big_int. The next minimum interval follows the same logic, sampling_array[I+2] - sampling_array[I+1] = Big_int and for all intervals after. Figure 5.1 shows a visual of how this weight is being sampled. The first sample of this weight (sample[I-1]) happens at cdf_weights[φ_I-1] + 1. The next sample happens (sample[I]) happens at cdf_weights[φ_I-1] + 2. After these first two sample, every sample after can be sampled at a minimum interval of Big_int. The maximum number of samples taken of this weight is equal to 6, which is equal to I_weights[φ_J]/Big_int + 2. The I_weights[φ_J]/Big_int part of the equation is a integer divide of the weight (I_weights[φ_J] = Big_int * 4 + 2), so there will be no decimal. Another way to think about the maximum is by thinking about it as a series of if statements: if a weight is greater than 2 then it could be sampled twice because the minimum sampling interval when given no information about the previous sample is 1. If a weight is greater than Big_int + 1 then it could be sampled 3 times because the minimum sampling interval, given that the last sample interval was 1, is Big_int (This means that sample[I+1] - sample[I-1] = Big_int + 1 which can fit inside of the weight because the weight is greater than Big_int + 1). Using these if statements and considering them for an arbitrary weight size allows us to conclude that the maximum number of samples taken of a weight is equal to I_weights[φ_J]/Big_int + 2. The minimum case follows the sample logic, we would set sampling_array[I] - sampling_array[I-1] = Big_int * 2 but this means that the next minimum interval is sampling_array[I+1] - sampling_array[I] = Big_int. The next minimum interval follows the same logic, sampling_array[I+2] - sampling_array[I+1] = Big_int and for all intervals after. Figure 5.1

shows a visual of how this weight is being sampled. The first sample of this weight (sample[I-1]) happens at cdf_weights[φ_I-1] + Big_int * 2. Every sample after can be sampled at a maximum interval of Big_int. The maximum number of samples taken of this weight is equal to 3, which is equal to I_weights[φ_J]/Big_int - 1. Now we have a range for the number of possible samples taken of a arbitrary weight (Eq 9 and 10): [I_weights[φ_J]/Big_int - 1, I_weights[φ_J]/Big_int + 2].

$$Min : \#ofsamples = \lfloor \frac{weights[\varphi_I]}{Big_int} \rfloor - 1 \tag{9}$$

$$Max : \#ofsamples = \lfloor \frac{weights[\varphi_I]}{Big_int} \rfloor + 2 \tag{10}$$

3.5 Formal SPARK Verification of Stratified Resampling

```
--ADB File:
function stratified_resampling (random_number_arr : I_Arr;
I_Weights : I_Arr)
return I_Arr
is
dup_arr : I_Arr := (others => 0);
cdf_weights : I_Arr := CDF(I_Weights);
sample : I_Arr := strat_samp_arr(random_number_arr);
begin
for I in sample'Range loop
if I > sample'First then
dup_arr (I) := dup_arr (I-1);
end if;

while sample(I) > cdf_weights(dup_arr(I)) loop
dup_arr (I) := dup_arr (I) + 1;
pragma Loop_Variant(Increases => dup_arr(I));--(I.1)

pragma Loop_Invariant(dup_arr(I) in cdf_weights'Range);--(I.2)

pragma Loop_Invariant(sample(sample'Last) <= cdf_weights(cdf_weights'Last));--(I.3)

pragma Loop_Invariant(if I = sample'First then dup_arr(I) > dup_arr'Loop_Entry(I) else dup_arr(I)
    > dup_arr(I-1));--(I.4)

pragma Loop_Invariant(sample(I) > cdf_weights(dup_arr(I)-1)
and then sample(I) <= cdf_weights(dup_arr(I)-1) + 2*big_int-1);--(I.5)
end loop;
```

```
28
29 pragma Loop_Invariant(for all K in dup_arr'Range => dup_arr(K) in cdf_weights'Range);--(O.1)
30
31 pragma Loop_Invariant(for all K in dup_arr'First+1..I => (for all J in dup_arr'First..K-1 =>
     dup_arr(K) >= dup_arr(J)));--(O.2)
32
33 pragma Loop_Invariant(if dup_arr(I) = 0 then sample(I) <= cdf_weights(dup_arr(I)) else
34 sample(I) > cdf_weights(dup_arr(I)-1) and then sample(I) <= cdf_weights(dup_arr(I)));--(O.3)
35
36 pragma Loop_Invariant(for all J in dup_arr'First..I =>
37 (if dup_arr(J) > 0 and then J = 0 then sample(J) - cdf_weights(dup_arr(J)-1) in 1 .. 2*big_int-1)
38 and then (if dup_arr(J) > 0 and then J > 0 and then dup_arr(J-1) /= dup_arr(J)
39 then sample(J) - cdf_weights(dup_arr(J)-1) in 1 .. 2*big_int-1));--(O.4)
40
41 pragma Loop_Invariant(for all K in sample'First..I =>
42 (for all J in sample'First..K =>
43 (if dup_arr(K) > 0 and then dup_arr(J) = dup_arr(K) then
44 sample(J) <= cdf_weights(dup_arr(K)) and then
45 sample(J) > cdf_weights(dup_arr(K)-1)) and then
46 (if dup_arr(K) = 0 and then dup_arr(J) = dup_arr(K) then
47 sample(J) <= cdf_weights(dup_arr(K)))));--(O.5)
48
49 pragma Loop_Invariant(for all K in dup_arr'First+1..I =>
50 (for all J in dup_arr'First..K =>
51 (if dup_arr(K-1) = dup_arr(J) and then dup_arr(K) /= dup_arr(J) then
52 (if J = 0 or (J > 0 and then dup_arr(J) > dup_arr(J-1)) then
53 K-J in I_Weights(dup_arr(J))/big_int - 1 ..
54 I_Weights(dup_arr(J))/big_int + 2)) and then
55
56 (if K = sample'Last and then dup_arr(K) = dup_arr(J) then
57 (if J = 0 or (J > 0 and then dup_arr(J) > dup_arr(J-1)) then
58 K+1-J in I_Weights(dup_arr(J))/big_int - 1 ..
59 I_Weights(dup_arr(J))/big_int + 2))));--(O.6)
60
61
62 end loop;
63
64 return dup_arr;
65 end stratified_resampling;
66
67
68 --ADS File:
69 function stratified_resampling (random_number_arr : I_Arr; I_Weights : I_Arr) return I_Arr with
70 Pre => I_Weights'First = 0 and then I_Weights'Last+1 <= max_particles and then
71 (for all K in random_number_arr'Range => random_number_arr (K) in 1 .. big_int) and then
72 (for all J in I_Weights'Range => I_Weights(J) in 0 .. big_int*I_Weights'Length) and then
73 I_Weights (I_Weights'First) = CDF(I_Weights)(I_Weights'First) and then
74 (for all J in I_Weights'Range => (if J = 0 then I_Weights(J) = CDF(I_Weights)(J)
75 else I_Weights(J) = CDF(I_Weights)(J) - CDF(I_Weights)(J-1))),
76
77 Post => (for all K in stratified_resampling'Result'First+1..stratified_resampling'Result'Last =>
78 (for all J in stratified_resampling'Result'First..K =>
79
80 (if stratified_resampling'Result(K-1) = stratified_resampling'Result(J) and then
```

```
81 stratified_resampling'Result(K) /= stratified_resampling'Result(J) then
82 (if J = 0 or (J > 0 and then stratified_resampling'Result(J) > stratified_resampling'Result(J-1))
       then
83 K-J in I_Weights(stratified_resampling'Result(J))/big_int - 1 ..
84 I_Weights(stratified_resampling'Result(J))/big_int + 2)) and then
85
86 (if K = stratified_resampling'Result'Last and then
87 stratified_resampling'Result(K) = stratified_resampling'Result(J) then
88 (if J = 0 or (J > 0 and then stratified_resampling'Result(J) > stratified_resampling'Result(J-1))
       then
89 K+1-J in I_Weights(stratified_resampling'Result(J))/big_int - 1 ..
90 I_Weights(stratified_resampling'Result(J))/big_int + 2))));
```

stratified_resampling Function

Similar to the SPARK verification of systematic resampling, we need to verify Eq (9) and (10), there are several loop invariants required. The nature of the outer for-loop in Appendix B is to iterate through samples with iterator I. The iterator, I, has a range between 0 and 499 because there are 500 particles. The inner while loop is responsible for iterating through cdf_weights until it finds an index where cdf_weights[φ_I] is greater than sample[I]. Both loops require descriptive loop invariants in order to prove our post condition that will be created from Eq (9) and (10). The only other piece of code in the algorithm that is not for the proof is setting the value of dup_arr[I] equal to dup_arr[I-1]; the reason for this is because we know that the next sample has to be at the same or a high index in the cdf_weights array. The only difference between the systematic and stratified resampling algorithm is the sampling array. Therefore, most of the loop invariants are the same with a couple of slight changes in the numbers.

The inner while loops first loop variant, (I.1), states that dup_arr[I] increases because we are iteratively searching for the index while in this loop, so it will always be increasing while inside. (I.2) is used to tell the compiler that the while loop will eventually exit, it does this by stating that the last element of the sample array is less than or equal to the last element of the cdf_weights array. The max value of Eq (10) is at the last element of the sampling array: 499*Big_int + Big_int (500*Big_int

is our integer representation of 1.0). The condition for being in the while loop is sample[I] is greater than cdf_weights[φ_I], this rule will eventually break as mentioned in (I.2). (I.3) makes sure that cdf_weights[φ_I] never gets an out of bound index. Since the most recent loop invariant guaranteed that we would exit the while loop we know that φ_I will never go out of bounds. Similar to how sample and cdf_weights elements get greater in value as we get farther into each of the arrays, φ has that very same property. All elements in φ will either be greater than or equal to the past values, (I.4) describes this characteristic of φ. Next we establish the relationship between sample[I] and cdf_weights[φ_I-1] with the last inner loop invariant, (I.5). As mentioned in the previous section, when the first sample of a weight is taken the value of sample[I] can be characterized in terms of cdf_weights[φ_I-1] to form a range: Sample[I] can be between cdf_weights[φ_I-1] + 1 and cdf_weights[φ_I-1] + Big_int*2 - 1. The reason that this loop invariant works inside of the inner whole loop but not the outer for loop is because if we are entering the inner while loop φ_I does not equal φ_I-1 (φ_I-1 and φ_{I-1} are different because φ_I-1 is the element before φ_I and φ_{I-1} is the element before φ_I that is inside of φ; φ_I could be equal to or greater than φ_I-1). If φ_I does not equal φ_I-1 then that means that it is the first sample being taken of cdf_weights[φ_I], when we are outside of the inner loop this is not guaranteed.

The purpose of the outer loop is to iterate through the sample array until we have taken as many samples as there are elements but now we need to prove that Eq (9) and (10) hold for each iteration. The (O.1) and (O.2) are there to make sure that (I.2) and (I.3) hold when in the outer loop. Even though there is no code that changes (I.2) and (I.3) in the outer loop, the compiler does not know this because it is in a for loop and we need a loop invariant to tell it what is going on. (O.3) relates directly to (I.5) but the upper bound has changed to include the case when φ_I equals φ_I-1; this means that it now includes the case where it is not the first sample taken

of cdf_weights[φ_I]. Therefore, in the outerloop all that can be said about the upper bound of sample[I] is that it is less than or equal to cdf_weights[φ_I]. The compiler needs to understand the relationship from (I.5) remains unchanged when iterating through the outer loop but, as we said in (O.3), (I.5) only occurs when it is the first sample being taken of a weight. (O.4) uses multiple if statements to desribe the relationship betwen sample and cdf_weights when it is the first sample being taken of cdf_weights and we are outside of the inner loop. When we are looking at the first iteration of the inner loop all that we have to check is that φ_I is greater than 0 because we are going to be using cdf_weights[φ_I-1]. When it is the first iteration of the outer loop (I) and φ_I is greater than 0, it means that sample[I] is greater than cdf_weights[φ_I-1] but less than cdf_weights[φ_I-1] + Big_int*2-1.The reason we have to specify this case is because φ_{I-1} does not exist yet which means that it will be the first sample being taken from cdf_weights[φ_I]. Now we have to account for the case where [φ_{I-1} does exist. When the iterator I is greater than 0, φ_I is greater than 0, and φ_I does not equal φ_I-1 then sample[I] is greater than cdf_weights[φ_I-1] but less than cdf_weights[φ_I-1] + Big_int*2-1. This is the same conditional statement as before but it checks that it is the first sample being taken from cdf_weights[φ_I]. (O.5) is a very important loop invariant because it characterizes the most unique trait of the φ array: when an arbitrary element φ_J equals another arbitrary element φ_K, every element in between the two is equal. (O.5) handles the case of when multiple samples are taken of the same weight. The outer for loop of the loop invariant, K, can range between 0 and I. The purpose of K is to act as an upper bound, where if the lower bound φ_J (the inner loop of the loop invariant) is equal to φ_K, everything in between is also equal. That does not mean that J and K are equal, for example, lets say that J is the second sample being taken in a sampling array and K is the 4 sample being taken but both are sampling the same element in the cdf_weights array (the elements

J and K are sampling is represented by φ_J and φ_K). The if statements inside of the for loops is similar to the logic of (O.4) because it handles when we are and are not on the first element of φ. When it is not the first element and φ_J equals φ_K there are two statements that can be said about the relationship between the sample and cdf_weights array: sample[J] is less than or equal to cdf_weights[φ_K] and sample[J] is greater than cdf_weights[φ_K-1]. The first statement is correct because φ_J equals φ_K and the inner while loop before the loop invariants makes sure that sample[J] is less than or equal to cdf_weights[φ_K]. The second statement is correct because we know that cdf_weights[φ_J] equals cdf_weights[φ_K] and sample[J] is greater than cdf_weights[φ_J-1], so when we interchange the two formulas we get that sample[J] is greater than cdf_weights[φ_K-1]. When it is the first element we can no longer use the index cdf_weights[φ_K-1] because that is out of bounds but we can say that sample[J] is less than or equal to cdf_weights[φ_K] for the same reasoning mentioned above.

The final loop invariant, (O.6), is what will be proving Eq (9) and (10). The loop invariants prior to (O.6) are all required in order for the compiler to understand it. In order to put Eq (9) and (10) in terms of code, two for loops are require. Similar to (O.5) the outer for loop, K, will serve as the upper bound and the inner for loop, J, will serve as the lower bound. We want to design the for loops so that the value of K-J is equal to the number of copies that are made of cdf_weights[φ_J]. In order to get K-J to equal the number of copies made of cdf_weights[φ_J] we need the first two if statements to be true: φ_J equals φ_K-1 and φ_K does not equal φ_J. This will make it so K-J equals the number of samples taken of a weight. Now that we have the number of copies we need to use it in Eq (9) and (10) which can be coded with the use of the PDF (the I_weights that were passed into the stratified resampling function). The final result is that K-J is in between the range of I_weights[φ_J]/Big_int - 1 and I_weights[φ_J]/Big_int + 2 which lines up directly to Eq (9) and (10). The next part of

the for loop is proving the exact same rule but it is for when we are at the end of the loop and φ_J equals φ_K. This is an edge case situation but it needs to be address as it changes the upper bound, in order to counter act this change we need to add 1 to K-J (the number of copies of element φ_J is now equal to K-J+1). The post condition for stratified resampling is a rewritten version of (O.6) as that is what we wanted to prove.

3.6 Additional Resampling Methods

```
--ADB File:
function bound_array (upper_bound : Integer; arr : I_Arr) return I_Arr is
n_arr : I_Arr (arr'Range);
begin
for J in arr'Range loop
if arr(J) >= upper_bound then
n_arr(J) = upper_bound;
else
n_arr(J) = arr(J);
end if;
pragma Loop_Invariant(for all K in arr'Range => n_arr(K) <= upper_bound);
end loop;
return n_arr;

end bound_array;

--ADS File:
function bound_array (upper_bound : Integer; arr : I_Arr) return I_Arr with
Post => for all K in arr'Range => bound_array'Result(K) <= upper_bound;
```

Explicit Proof

Residual and Multinomial are two other popularly used resampling methods. As described in the following sections, the proofs for both of these resampling methods provide nothing significant as the designed contracts would only be proving explicit relationships. The difference between proving explicit and implicit relationships in code is extremely important and needs to be understood in order to design meaningful contracts. The post conditions in systematic and stratified resampling are examples of implicit relationships because we are proving a rule that is not explicitly coded.

Implicit relationships are meaningful because we are proving a rule about an algorithm that describes what its final state should be. An example of a explicit proof can be shown in figure 6. The code in figure 6.1 takes in an integer array and an upper bound, where the algorithm will make sure that no number in the array goes above the upper bound and if it does it will hardcode it to the value of the upper bound. Figure 6.2 shows the post condition which iterates through the array and checks that no number is greater than the upper bound. This post condition is not significant because we explicitly wrote in our code that no number would go above the upper bound and, if it is, we will hardcode it to the value of the upper bound. Since the proofs that were designed for residual and multinomial fit the definition of being explicit we have deemed them to be insignificant.

3.6.1 Discussion of Multinomial Resampling Proof

```
function multinomial_resampling(random_number_arr : I_Arr;
I_Weights : I_Arr)
return I_Arr with
SPARK_Mode => Off
is
dup_arr : I_Arr := (others => 0);
cdf_weights : I_Arr := CDF(I_Weights);
sample : I_Arr := random_number_arr;
begin
for I in sample'Range loop
if I > sample'First then
dup_arr (I) := dup_arr (I-1);
end if;

while sample(I) > cdf_weights(dup_arr(I)) loop
dup_arr (I) := dup_arr (I) + 1;
end loop;

end loop;



return dup_arr;
end multinomial_resampling;
```

Multinomial Function

Multinomial resampling is a resampling method that is referenced and talked about but it is rarely implemented onto a system. The reason that multinomial resampling is rarely implemented is because, unlike systematic and stratified resampling which have the fixed interval in their sampling array on top of a random offset, multinomial resampling is completely random. Every index in the sampling array in multinomial is a random number between [1, Big_int * N], N being the number of particles. Since every index has a upper bound of Big_int * N all that can be said about the returning array of the resampling is that there are N indexes being returned and that every index is in [0, N]. Logically, multinomial resampling should still sample weights of higher magnitude but this is not guaranteed for every call to this method. The usefulness of the algorithm is not intuitive at first but it can be used in conjunction with other resampling methods, such as residual resampling. The code for multinomial resampling can be shown in figure 7. The code shown above is the same as systematic and residual resampling but the random_number_arr is a array of completely random numbers in the range of [1, Big_int * N].

3.6.2 Discussion of Residual Resampling Proof

```
function residual_resampling(random_number_arr : I_Arr;
Weights : F_Arr)
return I_Arr with
SPARK_Mode => Off
is
dup_arr : I_Arr := (others => 0);
min_copies : I_Arr := (others => 0);
residual : F_Arr := (others => 0.0);
I_Weights : I_Arr := (others => 0);
leftover : I_Arr;
idx : Integer := 0;
sum : Float := 0.0;
begin
for I in Weights'Range loop
min_copies(I) := Integer(Float'Truncation(Float(Weights'Length)*Weights(I)));
end loop;

for I in min_copies'Range loop
for J in 1 .. min_copies(I) loop
dup_arr(idx) := I;
```

```
idx := idx + 1;
end loop;
end loop;

for I in residual'Range loop
residual (I) := Weights(I) - Float(min_copies(I));
sum := sum + residual(I);
end loop;

for I in residual'Range loop
residual (I) := residual(I)/sum;
end loop;

I_Weights := int_weights(residual);
leftover := multinomial_resampling(random_number_arr, I_Weights);

while idx < dup_arr'Length loop
dup_arr(idx) := leftover(idx);
idx := idx + 1;
end loop;

return dup_arr;
end residual_resampling;
```

Residual Function

Residual resampling is the most complicated out of the four most popularly used resampling methods, in regards to the amount of code that is required. The idea behind residual resampling is that we want to guarantee a relationship between the magnitude of a particle's weight and the number of samples taken from it via an equation that is explicitly used in the code. The way that we guarantee a certain number of samples are taken of all the weights is by calculating the minimum number of samples we want for each particle (λ_I) and then filling our return array with the calculated number of indexes. After the first part of the return array has been initialized we will use another resampling to sample the rest of the particles needed, such as multinomial resampling. The code above shows the code for residual resampling, the first step is creating the min_copies array using Eq (11). In Eq (11) the weights array that we use is the same one that is passed into our int_weight function which means that the floating point error in this resampling method will not be corrected. Since

we will not be creating a CDF with the weights passed into the residual resampling function the floating point error is not significant enough to address. As a reminder, every index in the weights array follows Eq. (1.3), every index is between [0.0, 1.0], and the sum of the weights array is 1.0 (not including the floating point error). When considering these rules of the weights array we can understand the logic behind Eq (11): weights[I] * N can range between [0, N] which means that the maximum number of samples we can take of a particle is equal to N but that would only be in the case where we are 100% confident in a particle. The $\lfloor\rfloor$ operation truncates the float inside and converts it to an integer. Furthermore, we know that the sum of the weights array is equal to 1, so when we multiply each index by N and truncate each index we know that the sum of weights * N is less than or equal to N.

$$\lambda_I = \lfloor weights[I] * N \rfloor \tag{11}$$

Now that we have an array full of the minimum number of samples we want taken of each particle (λ), our next step is to fill our return array with the correct number of indices. We do this by using two for loops: the outside for loop iterates over λ and the inner for loop iterates from 1 to λ_I. Inside the for loops, dup_arr[idx] will be filled with the index I λ_I times. A quick example is shown below:

$\lambda_0 = 3$

$\lambda_1 = 2$

After for loops:

dup_arr = [0, 0, 0, 1, 1]

The counter, idx, not only serves as a placeholder for the index we are at in the

array but also tells us how many samples we have taken so far. In order to make use of idx we have to calculate the residual and generate another dup_arr from it. The way this is done is by using Eq (12).

$$residual[I] = weights[I] * N - Float(\lambda_I) \tag{12}$$

The residual array is initialized with the part of the float that was truncated when creating λ_I. Next we will normalize the residual array and pass it to the int_weights function to create a float point corrected weights array. Now we have everything we need to generate another dup_arr that is based on the magnitude of the residuals, we can use this to grab the N - idx particles we need to return a full dup_arr. In this version of residual resampling we decided to use multinomial resampling to sample the residual but any of the other three resampling methods mentioned in this paper would be viable.

Residual resampling is an effective resampling method but the idea behind it makes generating proofs for implicit relationships difficult. The interesting part about generating proofs for systematic and stratified resampling is that we were able to prove a mathematical relationship between the weights and the sampling array that guarantees the number of samples taken of a weight is between a specified range. In residual resampling there is no sampling array, instead we mathematically calculate the minimum number of samples we want of each particle without including any amount of randomness. The only random part of residual resampling is when we add in the N - idx samples that were generated from another resampling method. That being said, it does not mean that it is not possible to generate implicit proofs for residual resampling, it means that any other proofs that are generated are out of the scope of this paper because the purpose of this paper is to verify resampling methods via proofs about the relationship between the sampling and weights array.

IV. Application of Particle Filter

4.1 Preamble

This chapter presents analysis of our proofs and the results of a particle filter with our SPARK resampling methods used on satellite simulation data generated from Analytical Graphics systems tool kit (STK) software. The results will highlight the real world application of particle filters. In addition to the visuals presented, we shall also discuss any differences we found between commonly used python versions of our resampling methods that include some floating point error and our SPARK resampling methods.

4.2 Particle Filter Design

In order to show the application of a particle filter and how our Spark resampling method compares to commonly used python resampling methods we designed a simulation using satellite data from a systems tool kit (STK). The data that we receive through the STK will be used as the true state x $\in R^n$ at any particular time $t_i \in R^+$. In order to make a realistic simulation the particle filter will be used on a distribution of x that has noise added to it. The python file is divided into three main sections: SatSim, SatMod, Particle Filter. SatSim stores the x distribution and adds noise to all of the data in the distribution. SatMod is used by the particle filter to initialize and propagate data for creating the posterior and updating the distribution. Lastly, the particle filter is where we resample and change the weights of each particle.

4.2.1 SatSim

```
1 class SatelliteSim:
2 def __init__(self, Q=None, R=None, P0=None, project_loc="C:/Users/dorke/Downloads/PF_Resample/"):
3 #Setup the simulation timestep.
4 # Reading an excel file using Python
```

```
# Give the location of the file
loc = (project_loc + "pytest.xls")
wb = xlrd.open_workbook(loc)
sheet = wb.sheet_by_index(0)
sheet.cell_value(0, 0)
# init time steps
self.num_time_stps = sheet.nrows-1
self.time = np.zeros(sheet.nrows-1).reshape((sheet.nrows-1,1))
self.dt = np.zeros(sheet.nrows-2).reshape((sheet.nrows-2,1))
lat = np.zeros(sheet.nrows-1).reshape((sheet.nrows-1,1))
lon = np.zeros(sheet.nrows-1).reshape((sheet.nrows-1,1))
alt = np.zeros(sheet.nrows-1).reshape((sheet.nrows-1,1))
r_lat = np.zeros(sheet.nrows-1).reshape((sheet.nrows-1,1))
r_lon = np.zeros(sheet.nrows-1).reshape((sheet.nrows-1,1))
r_alt = np.zeros(sheet.nrows-1).reshape((sheet.nrows-1,1))
for i in range(sheet.nrows-1):
self.time[i] = sheet.cell_value(i+1, 12)
if i+2 < sheet.nrows:
self.dt[i] = sheet.cell_value(i+2, 12) - sheet.cell_value(i+1, 12)
lat[i] = sheet.cell_value(i+1, 13)
lon[i] = sheet.cell_value(i+1, 14)
alt[i] = sheet.cell_value(i+1, 15)
r_lat[i] = sheet.cell_value(i+1, 16)
r_lon[i] = sheet.cell_value(i+1, 17)
r_alt[i] = sheet.cell_value(i+1, 18)
self.store_state = np.concatenate((lat,lon,alt),axis=1)
self.inputs = np.concatenate((r_lat,r_lon,r_alt),axis=1)
# Init
self.dt_divider = np.ceil(10.*self.dt[0])
if Q is None:
#.2m, 3 degrees
self.Q = np.eye(3)*1000000**2
self.Q[2,2] = 100000000
else:
self.Q = Q
self.S_Q = la.cholesky(self.Q*self.dt[0]/self.dt_divider)
if R is None:
self.R = np.eye(3)*10000000000000
self.R[2,2] = 1000000
else:
assert R.shape == (3,3)
self.R = R
self.S_R = la.cholesky(self.R)
if P0 is None:
self.P0 = np.eye(3)
self.P0[2,2] = .0225
else:
self.P0 = P0
self.S_P0 = la.cholesky(self.P0)
self.store_meas = np.zeros(self.store_state.shape)
for idx in range(len(self.store_state)):
self.store_meas[idx] = self.store_state[idx] + self.S_R.dot(np.random.randn(3))
```

SatSim only contains a init file which initializes the data we will be using in the rest of this file. First we open up the spreadsheet and collect all of the data that the STK has stored inside of it. The state is then formed by concating the latitude, longitude, and altitude for each time step. In addition, we also initialize the inputs, which contains the velocity for the latitude, longitude, and altitude for each timestep (each timestep is also stored inside of the dt array). Next we initialize our Q, S_Q, R, S_R, P0, and SP_0 matrices: Q is the variance of the process noise, S_Q is the cholesky transformation of Q, R is the variance in our measurements, S_R is the cholesky transformation of R, P0 is the value of our Q matrix at time step 0, and S_P0 is our cholesky transformation of P_0. All of these matrices represent the knowledge we have of our system, we know that the data being returned from this satellite is not going to be perfect and that is the reason we have created these confidence matrices. In order to create a realistic set of data that we will be getting back from our satellite, we will add noise to the true data generated from the STK by using the S_R matrix dotted with an identity matrix containing three random numbers on the diagonal. The objective of this file is to use the particle filter to estimate our true location using the data with the noise added to it. We will know if the particle filter is working by graphing the particle filter estimation along with the true location of the satellite and watching the estimation converge to the true location.

4.2.2 SatMod

```
class SatelliteModel:
'''
'initialize' will generate a set of samples from P0
'propagate' will only move one step forward for the x values passed in
'p_z_given_x' returns p(z|x) for the z and x values passed in
'''
def __init__(self, Q=None, R=None, P0=None, dt=None):
if dt is None:
self.dt = .1
else:
self.dt = dt
```

```
#process noise
if Q is None:
self.Q = np.eye(3)*1000000**2
self.Q[2,2] = 100000000
else:
self.Q = Q
self.S_Q = la.cholesky(self.Q)
#measurement
if R is None:
self.R = np.eye(3)*1000000000000
self.R[2,2] = 100000
else:
assert R.shape == (3,3)
self.R = R
self.inv_R = la.inv(self.R)
self.gauss_const = 1./sqrt(la.det(2*pi*self.R))
if P0 is None:
self.P0 = np.eye(3)*10000000000000
self.P0[2,2] = 1000000
else:
self.P0 = P0
self.S_P0 = la.cholesky(self.P0)
def initialize(self, init_loc,num_samples):
'''
Sample from the initial PDF. Basically, assuming it is
a zero-mean Gaussian with covariance P0 (passed into __init__)
Args: num_samples: How many possible initializations you want
Returns: an num_samplesX3 numpy array
'''
return np.random.randn(num_samples,3).dot(self.S_P0.T) + init_loc
def propagate(self, state, inputs, dt, with_noise=True):
'''
Runs the satellite forward with inputs V and w _for one timestep_.
Args:
state: Can be either a (3,) numpy array or a (N,3) array
(Lat, Long, and Alt)
Returns:
An array the same size as state but propagated in time.
'''
my_state = state
if len(state.shape)==1:
my_state = my_state.reshape((1,3))
going_out = np.zeros(my_state.shape)
for i,x in enumerate(my_state):
dx = np.array([inputs[0], inputs[1], inputs[2]])*dt
going_out[i] = x + dx
if with_noise:
going_out[i] += self.S_Q.dot(np.random.randn(3))
if len(state.shape)==1:
going_out = going_out.reshape((3,))
return going_out
def p_z_given_x(self, z, x):
'''
Computes the probability of receiving the measurement z given
```

```
the value x
Args:
z: a (2,) numpy array holding the measurement
x: either a (3,) or (N,3) numpy array that holds either 1 or
multiple state values
Returns:
Either a scalar or a (N,) size numpy array with probabilities
'''
my_state=x
if len(x.shape)==1:
my_state = my_state.reshape((1,3))
going_out = np.zeros(len(my_state))
#Done data shaping. Compute the actual values
for i,particle_x in enumerate(my_state):
exponent = -0.5 * (z-particle_x).dot(self.inv_R.dot(z-particle_x))
going_out[i] = self.gauss_const * exp(exponent)
if len(x.shape)==1:
going_out = going_out[0].item()
return going_out
```

SatMod

The first function in SatMod, init, will initialize Q, S_Q, R, inv_R, gauss_const, P0, and S_P0. The second function in SatMod, initialize, will initialize a set number of particles from a zero-mean Gaussian distribution with covariance matrix P0. Each of our particles will represent a location in space where we might be (each particle will have a 1 by 3 matrix associated with it to represent the latitude, longitude, and altitude of that particle). Each particle will also have a weight assigned to it that represents the confidence we have that the particle is our true location. The arguments we need for this function is the initial location from the STK data and then the number of samples. The third function in SatMod, propagate, will propagate our state forward using the inputs that were passed into the function, along with the timestep. This is done by multiplying the inputs (our velocity in each direction) by the change in time, dt. This array, dx, is then added to our state, x, along with some process noise. The last function in SatMod, p_z_given_x, calculates the probability of receiving the measurement z given the value of each state, x. This is done by using our particles array, x, and our new measurement z (all of our measurements come from the

data array that was made in SatSim, store_meas) as inputs to Eq (13)–The equation for a normal distribution bell curve. The purpose of this function is important to understand, the p_z_given_x array is the probability of getting measurement z given our location is state x for every particle that we have and it is the main calculation we have to make for reweighting each particle.

$$p_z_given_x[I] = \frac{1}{\sqrt{det(2\pi R)}} e^{\frac{-(z - x[I]) \cdot inv_R \cdot (z - x[I])}{2}} \tag{13}$$

4.2.3 Particle_Filter

```
class ParticleFilter:
def __init__(self,model, init_loc ,N=500,resample_threshold=200, method=None):
self.particles = model.initialize(init_loc, N)
self.weights = np.ones(N)/float(N)
self.N = N
self.resample_threshold = resample_threshold
#Used for systematic resampling
self.steps = np.ones(N)/float(N)
self.steps[0] = 0
self.steps = np.cumsum(self.steps)
self.resampling_method = method
self.model = model
def compute_Neff(self):
return 1/np.sum(np.square(self.weights))
def resample(self):
# python code for systematic resampling
if self.resampling_method == "Systematic":
random_num = np.loadtxt("random.out",dtype= float, unpack=False)
cdf_weights = np.cumsum(self.weights)
sampling_arr = random_num/float(self.N) + self.steps
py_idxs = np.searchsorted(cdf_weights, sampling_arr)
elif self.resampling_method == "Stratified":
random_arr = np.loadtxt("random.out",dtype= float, unpack=False)
cdf_weights = np.cumsum(self.weights)
sampling_arr = random_arr/float(self.N) + self.steps
py_idxs = np.searchsorted(cdf_weights, sampling_arr)
# Ada
ada_idxs = np.loadtxt("Output.out",dtype= int, unpack=False)
# Check Ada vs Python
check = np.concatenate((py_idxs.reshape(500,1), ada_idxs.reshape(500,1)), axis=1)
np.savetxt("check.out", check, fmt='%i, %i', header="Python, Ada")
# Update particles and reset weights
self.particles = self.particles[ada_idxs]
self.weights = np.ones(self.N)/float(self.N)
def step(self, meas, inputs,dt, project_loc):
#Check if we need to resample and resample
```

```
if self.compute_Neff() < self.resample_threshold:
np.savetxt("Input.out", pf.weights, fmt='%.18f')
os.system(project_loc + "obj/main " + self.resampling_method)
self.resample()
open('Output.out', 'w').close()
open('Input.out', 'w').close()
#propagate forward if needed
self.particles = self.model.propagate(self.particles,inputs,dt)
#Apply the measurement
self.weights *= self.model.p_z_given_x(meas,self.particles)
self.weights /= np.sum(self.weights)
def get_average(self):
return np.average(self.particles,axis=0,weights=self.weights)
```

Particle_Filter

The init function in the particle filter class initializes the particles, weights, N, resample threshold, steps array, resampling method, and model. The particles array is initialized using the initialize function from SatMod to randomly sample the state of each particle from a gaussian distribution. The weight of each particle will always initially be $\frac{1}{N}$ because we have no knowledge of p_z_given_x, so every particle is weighted the same. N is set to the amount of particles that we want, in our simulation and throughout all of our proofs N is set to 500 particles. The resampling threshold is, like the name suggests, the threshold at which if the number of effective particles is less than we will resample. The next function, compute_Neff in the particle filter, calculates the number of effective particles via Eq (14).

$$Neff = \frac{1}{sum(weights^2)} \tag{14}$$

The resample function calls our function in Spark and gathers all of the data that it outputs along with using a commonly used python resampling method to generate another version of dup_arr. The reason that we generate two versions of dup_arr is because we want to show that our float point correct version of a resampling method generates the same results as one with the floating point error included. Since we are only using 500 particles there will not be much of a difference between the two arrays

as the error that is generated from creating the CDF does not get big enough to create a significant error. Furthermore, our main reason for converting the float weights array to an integer weight array is because we needed to since we were making proofs (proofs can become very complicated if we are working with floats). We call the Spark verified function by using a system command and then grabbing the data from an output file that was generated by our Spark function. Our dup_arr function is then used as a mask to create the new particles array (self.particles = self.particles[dup_arr]). Lastly, we reset our weights to be $\frac{1}{N}$. The step function in the particle filter class will first check if we need to resample and, if we do, it will call the resampling function. Next, the step function will propagate all of the particles one time step by using the SatMod propagate function. Lastly, we will calculate the new weights by multiplying the new function by p_z_given_x and then normalizing the weights array.

4.2.4 Additional Methods

```
def plot_states(states, meas=None):
fig = plt.figure()
ax = fig.add_subplot(projection='3d')
ax.plot(states[:,0], states[:,1], states[:,2], label='Satellite', c='b')
if meas is not None:
ax.scatter(meas[:,0], meas[:,1], meas[:,2],marker='.', label='Measurements', c='r')
ax.set_xlabel('Lat Label')
ax.set_ylabel('Lon Label')
ax.set_zlabel('Alt Label')
plt.show()
def weight_to_size(weights, max_size=100, min_size=1):
'''
This function attempts to take a set of weights from the particles
for a particle filter and turn them into a set of sizes. This allows
matplotlib's scatter to plot the particles with different sizes
Args:
weights: The weights on the samples to be graphed
max_size: The size of the biggest dot that will be graphed
min_size: The size of the smallest dot that will be graphed
Returns:
A numpy array of the same size as weights with sizes that can be passed
in as the "size" parameter into plt.scatter
'''
# Let's try a linear scaling from weights to size...
max_w = np.max(weights)
min_w = np.min(weights)
```

```
if max_w==min_w:
return np.ones(len(weights))*10
else:
return np.rint(weights * (max_size-min_size)/(max_w-min_w) + min_size).astype(int)
def plot_particles(ax, particles, weights, truth_hist, time):
ax.plot(truth_hist[:,0], truth_hist[:,1], truth_hist[:,2], label='Satellite')
sizes = weight_to_size(weights)
ax.scatter(particles[:,0], particles[:,1], particles[:,2], marker='o', label='Measurements', c='g'
    ,s=sizes)
plt.title(f'time is {time}seconds')
plt.show()
if __name__ == "__main__":
loc = "C:/Users/dorke/Downloads/PF_Resample/"
methods = ["Stratified", "Systematic"]
Resample_Method = methods[0]
us = SatelliteSim(project_loc = loc)
plot_states(us.store_state, us.store_meas)
#Now let's run the estimator
um = SatelliteModel(dt=us.dt)
pf= ParticleFilter(um,us.store_meas[0,:], N=500,method=Resample_Method)
#Now run the particle filter
for i in range(us.num_time_stps-1):
if i%10 == 0:
#plot the state...
fig = plt.figure()
ax = fig.add_subplot(projection='3d')
plot_particles(ax, pf.particles, pf.weights, us.store_state[:i+1], us.time[i])
#plt.plot(pf.weights)
#plt.show()
pf.step(us.store_meas[i+1], us.inputs[i], us.dt[i], loc)
```

Additional Methods

The plot_states function will plot all of our states on top of the true states from our SatSim. Weight_to_size will make sure that the plot_particles function has different sizes based on the magnitude of each particle's weight. The plot_particles function will plot each one of our particles on top of the true states from our SatSim. Lastly, our main method will initialize our SatSim, SatMod, and particle filter. The main method will then call the particle filter step function and the plot_particles function inside of a for loop that ends when we have iterated over every timestep.

```
Summary of SPARK analysis
=========================

-----------------------------------------------------------------------------------------------------------------
SPARK Analysis results        Total      Flow   CodePeer                                Provers   Justified   Unproved
-----------------------------------------------------------------------------------------------------------------
Data Dependencies                 .         .          .                                      .           .          .
Flow Dependencies                 .         .          .                                      .           .          .
Initialization                    2         2          .                                      .           .          .
Non-Aliasing                      .         .          .                                      .           .          .
Run-time Checks                 239         .          .                  239 (CVC4 73%, Z3 27%)          .          .
Assertions                       66         .          .  66 (CVC4 55%, Trivial 17%, Z3 28%)          .          .
Functional Contracts             24         .          .  24 (CVC4 79%, Trivial 11%, Z3 11%)          .          .
LSP Verification                  .         .          .                                      .           .          .
Termination                       2         .          .                          2 (CVC4)            .          .
Concurrency                       .         .          .                                      .           .          .
-----------------------------------------------------------------------------------------------------------------
Total                           333   2 (1%)          .                        331 (99%)             .          .


max steps used for successful proof: 9126

Analyzed 3 units
in unit helper_methods, 6 subprograms and packages out of 6 analyzed
  helper_methods at helper_methods.ads:1 flow analyzed (0 errors, 0 checks and 0 warnings) and proved (0 checks)
  helper_methods.CDF at helper_methods.ads:31 flow analyzed (0 errors, 0 checks and 0 warnings) and proved (11 checks)
  helper_methods.PDF at helper_methods.ads:39 flow analyzed (0 errors, 0 checks and 0 warnings) and proved (22 checks)
  helper_methods.int_weights at helper_methods.ads:47 flow analyzed (0 errors, 0 checks and 0 warnings) and proved (6 checks)
  helper_methods.strat_samp_arr at helper_methods.ads:19 flow analyzed (0 errors, 0 checks and 0 warnings) and proved (41 checks)
  helper_methods.sys_samp_arr at helper_methods.ads:11 flow analyzed (0 errors, 0 checks and 0 warnings) and proved (13 checks)
in unit main, 0 subprograms and packages out of 1 analyzed
  Main at main.adb:8 skipped
in unit resample, 3 subprograms and packages out of 5 analyzed
  resample at resample.ads:3 flow analyzed (0 errors, 0 checks and 0 warnings) and proved (0 checks)
  resample.multinomial_resampling at resample.ads:53 skipped
  resample.residual_resampling at resample.ads:55 skipped
  resample.stratified_resampling at resample.ads:30 flow analyzed (0 errors, 0 checks and 0 warnings) and proved (119 checks)
  resample.systematic_resampling at resample.ads:7 flow analyzed (0 errors, 0 checks and 0 warnings) and proved (119 checks)
```

Figure 10: Proof Summary

4.3 Proof Analysis

Figure 10 shows the SPARK summary of our proof. There are a total of 333 successful proofs in our code. No proof was left unchecked and every method used in our code, aside from the main method which was used to communicate between python and SPARK methods, had zero errors or warnings. The grade achieved by each resampling method that we verified was gold. SPARK has 5 different ways of categorizing proofs: stone, bronze, gold, and platinum. Stone means that the proof adheres to the SPARK subset, Bronze means that it was also able to check for references to uninitialized variables, Silver means that it was also absent of any runtime errors, Gold means that it also kept integrity of properties inside of the proofs, and Platinum means that it also had full functional correctness. In summary, our proof was successful and we were able to verify a crucial mathematical relationship

that is required for each resampling method to work every iteration.

4.4 Particle Filter Satellite Simulation

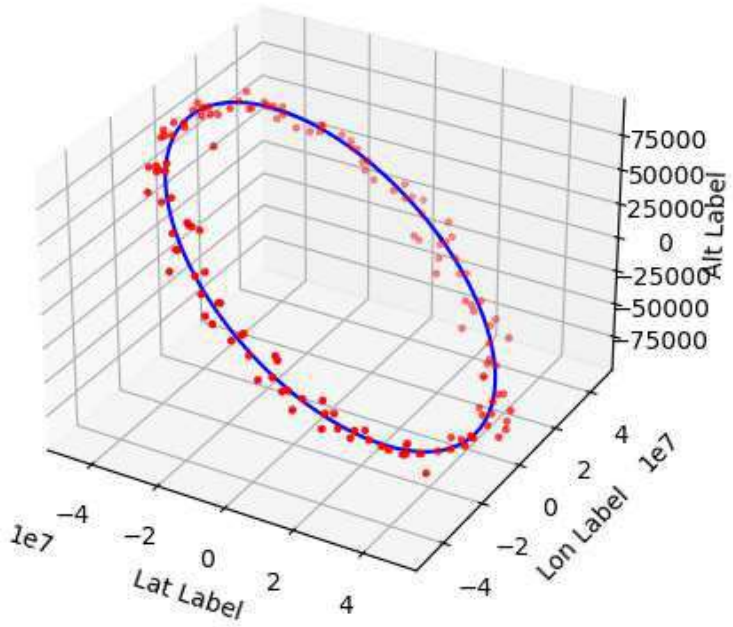
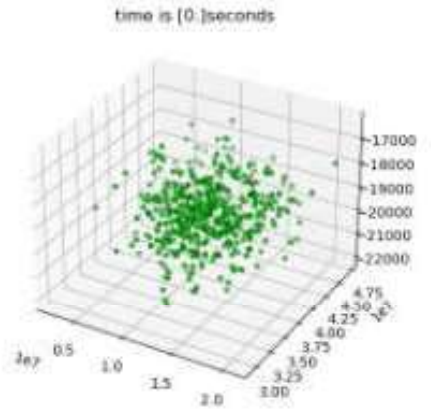
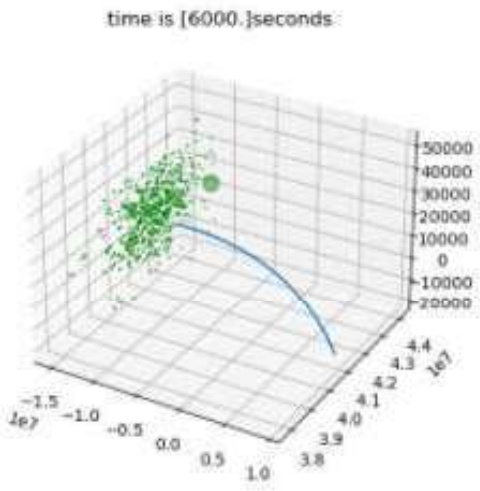
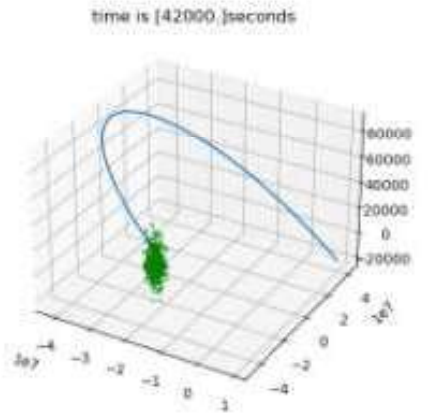


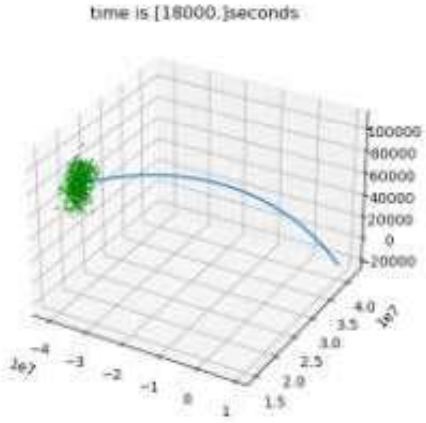
Figure 11: True Location and Measurements of Satellite

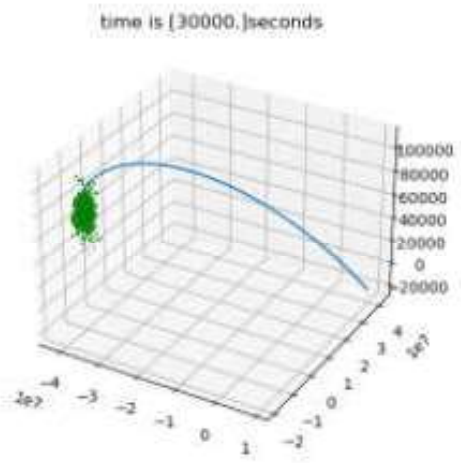
The true orbit of the satellite is represented by the blue circle in figure 11, this data was generated through the STK software mentioned earlier. The measurements are represented by the red dots that surround the orbit. Figure 10 shows all of the data at once but we will be taking this data and iteratively passing the measurements through the particle filter. The particle filter will use these measurements to estimate the true location, and at the end of the simulation the particle filter should have produced an orbit similar to the blue orbit shown in figure 10. As shown in the code for this simulation, we will be using our resampling method from SPARK to resample all of the particles.

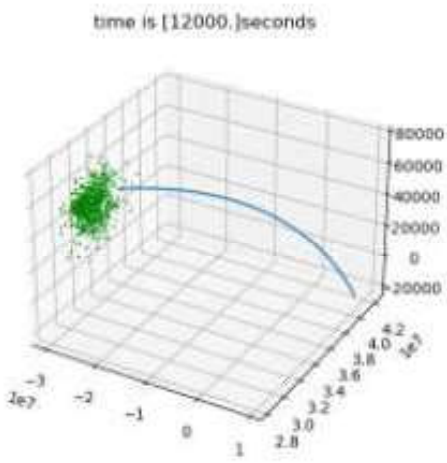

time is [0.]seconds


time is [6000.]seconds


time is [42000.]seconds


time is [18000.]seconds


time is [30000.]seconds


time is [12000.]seconds

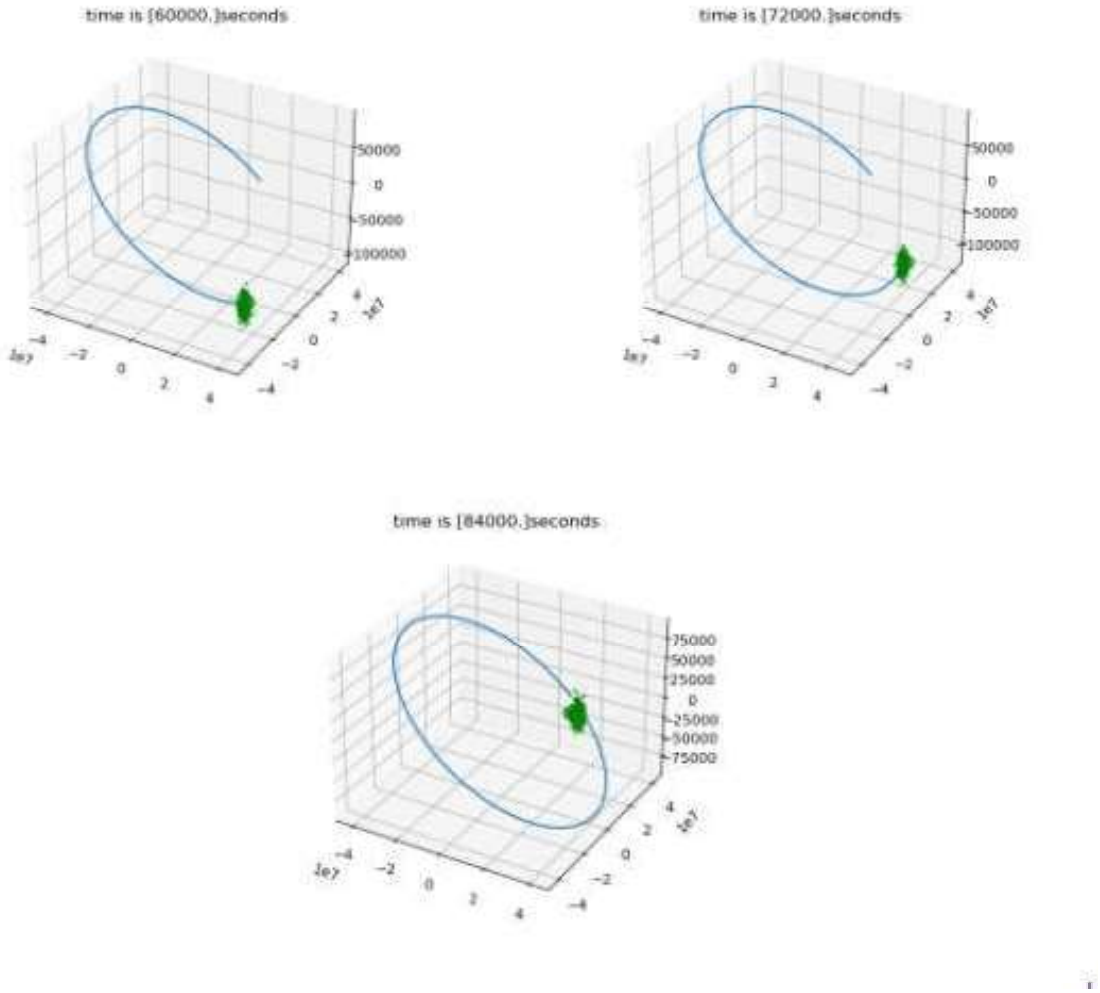
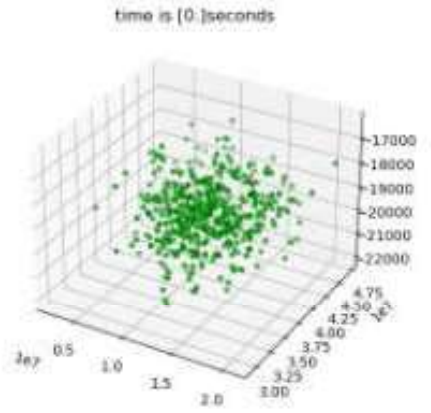
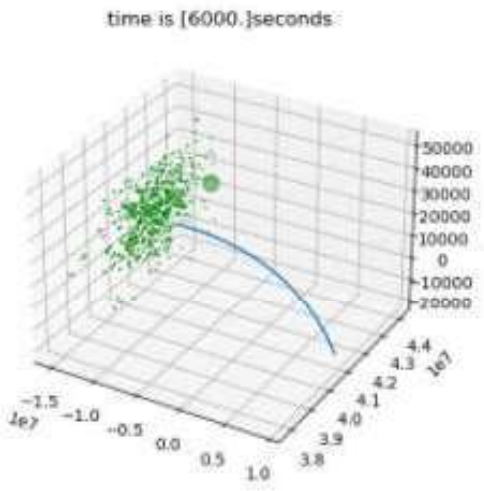


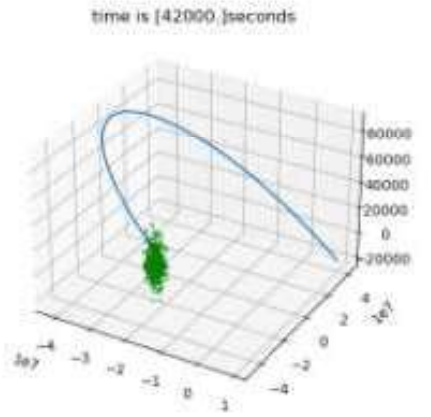
Figure 12: Systematic Timesteps

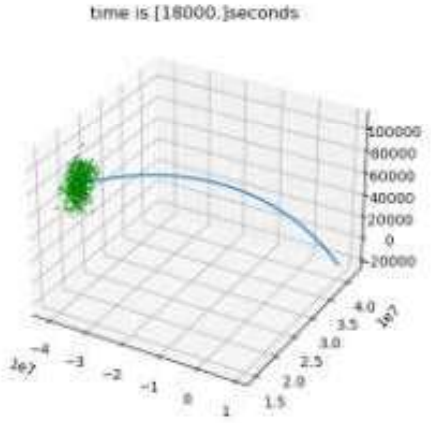
Figure 12 shows the development of the true satellite orbit with the use of a particle filter–the latitude, longitude, and altitude have all been converted to miles. Each particle is represented by a green dot, there will always be 500 particles as that is the amount we decided on using for our particle filter. As shown at time = 0 seconds, the particles have no information on where the true location is because we have not passed the measurements in yet. Once we do pass the measurements in the particle location changes to where we think the true location of the satellite is at time = 6000 seconds. As we progress through all of the timesteps we can see that the particles start to converge to a concentrated area that is much smaller than the one

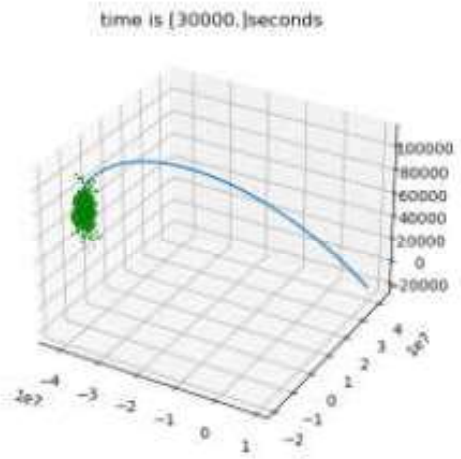
at time = 0 seconds. The reason for this convergence is because of our systematic resampling method.

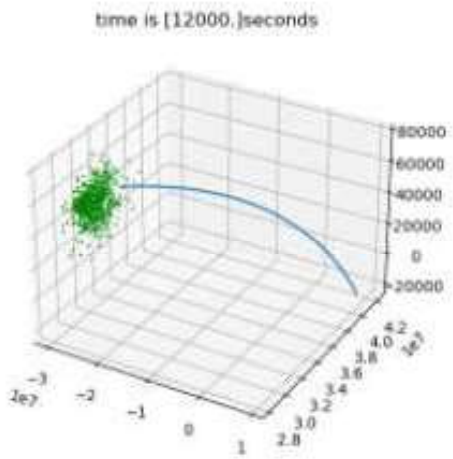

time is [0.]seconds


time is [6000.]seconds


time is [42000.]seconds


time is [18000.]seconds


time is [30000.]seconds


time is [12000.]seconds

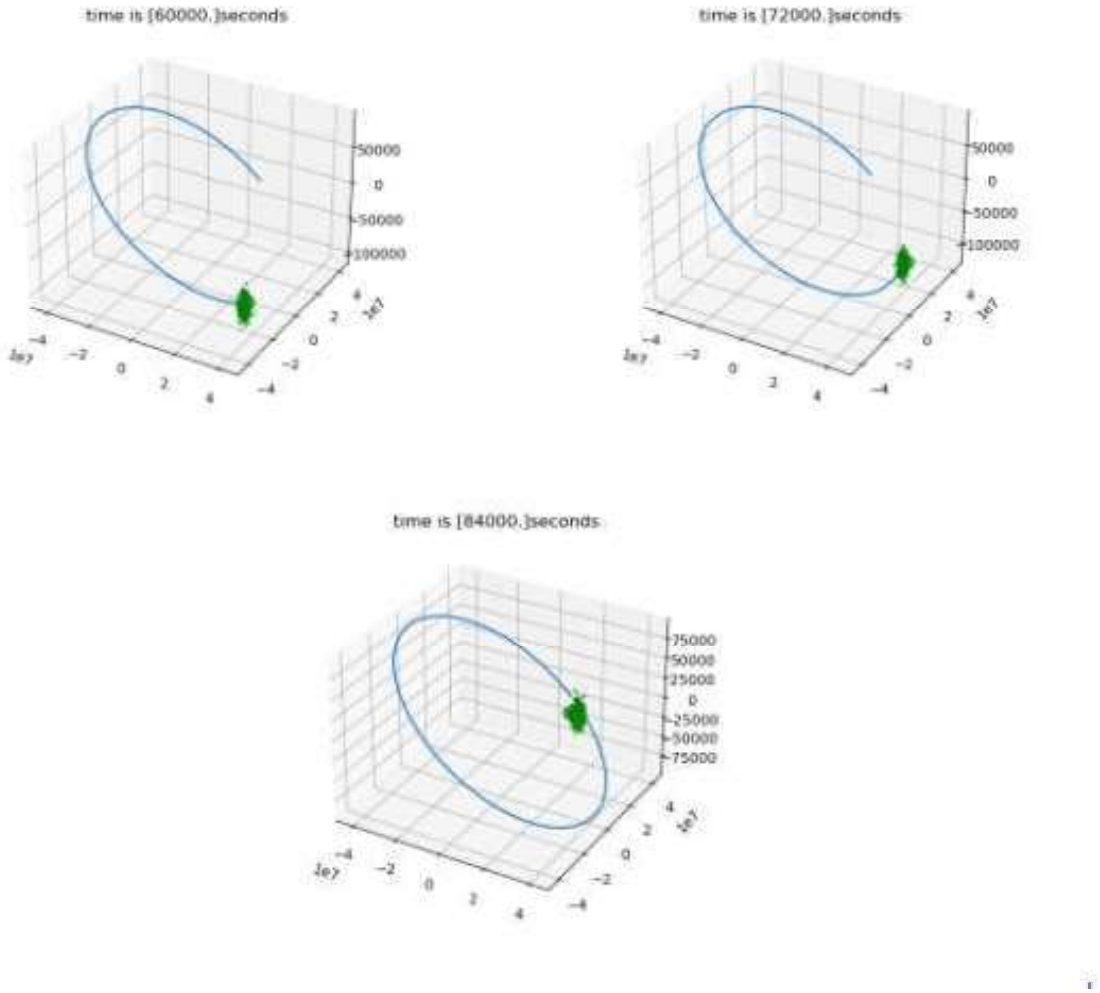


Figure 13: Stratified Timesteps

Figure 13 shows the development of the true satellite orbit with the use of the same particle filter as figure 12 but this time we are using the stratified resampling method we made. This resampling method produces the same results, and thus accomplishes the same goal as our systematic resampling method. When we are only using 500 particles and our measurement variance is relatively low, the output results of each resampling method become very similar. Our objective for these figures was to show how a particle filter works and whether our two verified resampling methods work as they should. As shown above, our particle filter successfully estimated the true orbit of the satellite, along with giving a visual representation of what each of our

resampling methods do.

V. Conclusions

This paper discusses the logic behind various resampling methods and proves key characteristics about the algorithms through the use of SPARK. The algorithms for systematic and stratified resampling were simple to implement but the relationships that were created between the sampling and particle weights array turned out to be considerably more complex than anticipated. For systematic resampling, we logically deduced that a major characteristic of the algorithm is that it will sample any weight that is greater than the fixed interval in the sampling array at least once. We were able to prove a more generalized version of this characteristic by creating a range for the number of possible samples taken of a weight based on the size. In systematic resampling, we found that the minimum number of samples taken of a weight is the number of times that the fixed interval can fit inside of it, and that the maximum number of samples taken is the minimum plus one [52, 53, 54]. For stratified resampling, we found that the relationships between the sampling and particle weights array were conceptually the same but since the structure of the sampling array changed the range for the possible number of samples for a weight was also changed. The key characteristic of the stratified sampling array is that the size between past samples affects the range for the current samples being taken. Furthermore, this difference between the systematic and stratified sampling array cause the range for the possible number of samples for a specific weight to change: the minimum number of samples taken for a weight is the number of times a fixed interval can fit inside of the weight minus one and the maximum number of samples taken for a weight is the number of times a fixed interval can fit inside of the weight plus two [55, 56, 57]. Proving the characteristics mentioned above allowed us to determine that the resampling algorithms were working. When we tried to find and prove important characteristics of multinomial and residual resampling we found that both of these algorithms showed

very little significance in proving. For multinomial resampling, the sampling array is completely random so there is nothing that will stay consistent for every use of this algorithm. For residual resampling, the sampling array was designed in a way where there is no randomness for calculating the minimum number of samples taken of a specific weight. While determining the minimum number of samples in residual resampling is different than in systematic and stratified resampling the end result is the same. The reason that we found it difficult to make a significant proof about residual resampling is because residual resampling uses multinomial resampling for generating the remaining samples, which also determines the maximum for the number of possible samples. Since the maximum for residual resampling is completely random, the only significant characteristic to prove is the minimum value for the number of possible samples taken of a specific weight. As mentioned before, the minimum for the number of possible samples taken of a specific weight is mathematically determined with a simple formula and hard coded to the calculated value–this shows very little significance in proving [58, 59]. Lastly, this paper showed through a particle filter simulation that the integer domain version of our resampling methods is interchangeable with the floating point domain versions. The simulation also showed a visual representation of our verified resampling methods being used on satellite data.

5.1 Limitations

SPARK has been an incredibly helpful tool in developing the proofs of systematic and stratified resampling but it does come with several limitations [60]. SPARK is able to achieve many of its proofs with its limited language features but if many of these resitrication were to be lifted it would allow for more advanced proofs [61]. These limitations became very apparent when we were first designing our CDF proofs and found that it would be extremely difficult to design them in the floating point

domain. It has been mentioned by the developers of SPARK that future releases may lift some of these restrictions [62, 63].

5.2 Future Work

The next steps that build upon this work are as follows:

- The residual resampling that is discussed in this paper could be changed to use any of the other three resampling methods when sampling from the residual. If the residual were to be sampled by systematic or stratified resampling it would allow a numerical maximum to be generated. In addition to changing the way that residual resampling is implemented, a future project could focus on proving relationships that are not derived from determining the range for the possible number of samples for a specific weight.

- The proof for stratified resampling can be expanded so that it recursively incorporates all of the past sizes between each sample to narrow down the range even more than it is when using the most recent difference between two samples.

- This entire could be done in the floating point domain but each proof would have to account for the floating point error. This could be done with a more advanced version of SPARK.

Appendix A. Helper_Methods

```
package body helper_methods with
SPARK_Mode
is

   function sys_samp_arr
     (offset : Integer)
      return I_Arr
   is
      steps : I_Arr;
   begin
      steps := (steps'First..steps'Last => 0);
      for I in steps'First .. steps'Last loop
         steps (I) :=  I*big_int + offset;
         pragma Loop_Invariant(for all J in steps'First .. I => steps (J) = J*big_int + offset);

         pragma Loop_Invariant(for all J in steps'First+1 .. I =>
                                  (for all K in steps'First..J-1 => steps(J) > steps(K)));
      end loop;
      return steps;
   end sys_samp_arr;

   function strat_samp_arr
     (offset_arr : I_Arr)
      return I_Arr
   is
      steps : I_Arr;
   begin
      steps := (steps'First..steps'Last => 0);
      for I in steps'First .. steps'Last loop
         steps (I) :=  I*big_int + offset_arr (I);
         pragma Loop_Invariant(for all J in steps'First .. I => steps (J) = J*big_int + offset_arr
     (J));

         pragma Loop_Invariant(for all K in steps'First .. I =>
                                  (if K = steps'First then steps(K) in 1 .. big_int
                                   else steps(K) in K*big_int + 1 .. K*big_int + big_int));

         pragma Loop_Invariant(for all J in steps'First+1 .. I =>
                                  (for all K in steps'First .. J-1 =>
                                     steps (J) - steps(K) in (J-K-1)*big_int+1 .. (J-K+1)*(big_int)
     -1));

         pragma Loop_Invariant(for all K in steps'First+1 .. I =>
                                  (if offset_arr(K-1) = 1 and then offset_arr(K) = big_int
                                   then steps(K) - steps(K-1) = 2*big_int-1));
      end loop;
      return steps;
   end strat_samp_arr;


   function CDF
     (A : I_Arr)
```

```
51       return I_Arr
52    is
53       Result : I_Arr := (A'First..A'Last => 0);
54    begin
55
56       Result (A'First) := A(A'First);
57       for I in A'First + 1 .. A'Last loop
58          Result (I) := Result (I-1) + A (I);
59          if Result (I) > big_int * A'Length then
60             Result (I) := big_int * A'Length;
61          elsif Result (I) < 0 then
62             Result (I) := 0;
63          end if;
64          pragma Loop_Invariant(Result (A'First) = A(A'First));
65
66          pragma Loop_Invariant(for all K in Result'Range => Result (K) >= 0 and
67                                   Result (K) <= big_int * A'Length);
68
69          pragma Loop_Invariant(for all K in Result'First .. I =>
70                                   (for all J in Result'First .. K => Result (K) >= Result (J)));
71       end loop;
72       Result(Result'Last) := big_int * A'Length;
73       return Result;
74    end CDF;
75
76    function PDF
77      (A : I_Arr)
78       return I_Arr
79    is
80       CDF_A : I_Arr := CDF(A);
81       Result : I_Arr := (A'First..A'Last => 0);
82    begin
83
84       Result (A'First) := A(A'First);
85       for I in Result'Range loop
86          if I = 0 then
87             Result(I) := CDF_A(I);
88          else
89             Result(I) := CDF_A(I) - CDF_A(I-1);
90             end if;
91          pragma Loop_Invariant(Result (A'First) = A(A'First));
92
93          pragma Loop_Invariant(for all K in Result'Range => Result (K) >= 0 and
94                                   Result (K) <= big_int * A'Length);
95
96          pragma Loop_Invariant(for all K in Result'First .. I =>
97                                   (if K = 0 then Result(K) = CDF_A(K) else Result(K) = CDF_A(K) -
     CDF_A(K-1)));
98       end loop;
99       return Result;
100   end PDF;
101
102   function int_weights
103     (A : F_Arr)
```

```
104       return I_Arr
105    is
106       int_weights_arr : I_Arr := (A'First..A'Last => 0);
107    begin
108       for I in A'Range loop
109          int_weights_arr (I) := Integer(Float(A'Length * big_int) * A(I));
110          if int_weights_arr (I) > big_int * A'Length then
111             int_weights_arr (I) := big_int * A'Length;
112          elsif int_weights_arr (I) < 0 then
113             int_weights_arr (I) := 0;
114          end if;
115          pragma Loop_Invariant (for all K in int_weights_arr'First .. I =>
116                                    int_weights_arr (K) in 0 .. big_int * A'Length);
117       end loop;
118       return int_weights_arr;
119    end int_weights;
120
121
122 end helper_methods;
```

Listing A.1: helper_methods.adb

```
1  package helper_methods is
2     pragma SPARK_Mode;
3     type F_Arr is array (0 .. 499) of Float;
4     type I_Arr is array (0 .. 499) of Integer;
5     big_int : constant Integer := 2000000; -- 1E7
6     max_particles : constant Integer := 500;
7     subtype Positive is Integer range 0 .. big_int*max_particles;
8     subtype rand_range is Integer range 1 .. big_int;
9
10    function sys_samp_arr (offset : Integer) return I_Arr with
11      Pre =>  offset >= 1 and then offset <= big_int,
12      Post => sys_samp_arr'Result (sys_samp_arr'Result'First) = offset and then
13      (for all J in sys_samp_arr'Result'Range => sys_samp_arr'Result (J) = J*big_int + offset) and
      then
14      sys_samp_arr'Result (sys_samp_arr'Result'Last) <= sys_samp_arr'Result'Length*big_int and then
15      (for all J in sys_samp_arr'Result'First+1 .. sys_samp_arr'Result'Last =>
16         (for all K in sys_samp_arr'Result'First..J-1 => sys_samp_arr'Result(J) > sys_samp_arr'
      Result(K)));
17
18    function strat_samp_arr (offset_arr : I_Arr) return I_Arr with
19      Pre =>  (for all K in offset_arr'Range => offset_arr (K) in 1 .. big_int),
20      Post => strat_samp_arr'Result (strat_samp_arr'Result'First) = offset_arr(offset_arr'first)
      and then
21      strat_samp_arr'Result (strat_samp_arr'Result'Last) <= strat_samp_arr'Result'Length*big_int
      and then
22      (for all J in strat_samp_arr'Result'Range => strat_samp_arr'Result (J) = J*big_int +
      offset_arr(J)) and then
23      (for all K in strat_samp_arr'Result'Range =>
24         (if K = strat_samp_arr'Result'First then strat_samp_arr'Result(K) in 1 .. big_int
25              else strat_samp_arr'Result(K) in K*big_int + 1 .. K*big_int + big_int)) and then
26      (for all J in strat_samp_arr'Result'First+1 .. strat_samp_arr'Result'Last =>
27         (for all K in strat_samp_arr'Result'First .. J-1 =>
```

```
28              strat_samp_arr'Result (J) - strat_samp_arr'Result (K) in (J-K-1)*big_int+1 .. (J-K+1)
    *(big_int)-1));
29
30    function CDF (A : I_Arr) return I_Arr with
31      Pre => (for all J in A'Range => A(J) <= big_int*A'Length and A(J) >= 0),
32      Post => A (A'First) = CDF'Result(CDF'Result'First) and then
33      CDF'Result(CDF'Result'Last) = big_int*A'Length and then
34      (for all I in CDF'Result'Range => CDF'Result (I) >= 0 and then
35           CDF'Result (I) <= big_int*A'Length) and then
36      (for all K in CDF'Result'Range => (for all J in CDF'Result'First .. K => CDF'Result (K) >=
    CDF'Result (J)));
37
38    function PDF (A : I_Arr) return I_Arr with
39      Pre => (for all J in A'Range => A(J) <= big_int*A'Length and A(J) >= 0),
40      Post => A (A'First) = CDF(A)(CDF(A)'First) and then
41      (for all I in PDF'Result'Range => PDF'Result (I) >= 0 and then
42           PDF'Result (I) <= big_int*A'Length) and then
43      (for all K in PDF'Result'Range => (if K = 0 then PDF'Result(K) = CDF(A)(K)
44                                         else PDF'Result(K) = CDF(A)(K) - CDF(A)(K-1)));
45
46    function int_weights(A : F_Arr) return I_Arr with
47      Pre => (for all I in A'Range => A(I) in 0.0 .. 1.0),
48      Post => (for all J in int_weights'Result'Range => int_weights'Result (J) <= big_int*A'Length
    and then
49           int_weights'Result (J) >= 0);
50
51  end helper_methods;
```

Listing A.2: helper_methods.ads

Appendix A. Resample

```
with helper_methods; use helper_methods;
with resample; use resample;

package body resample with
SPARK_Mode => ON
is

   function systematic_resampling (random_number : Integer;
                                   I_Weights : I_Arr)
                                   return I_Arr
   is
      dup_arr : I_Arr := (others => 0);
      cdf_weights : I_Arr := CDF(I_Weights);
      sample : I_Arr := sys_samp_arr(random_number);
   begin

      for I in sample'Range loop
         if I > sample'First then
            dup_arr (I) := dup_arr (I-1);
         end if;

         while sample(I) > cdf_weights(dup_arr(I)) loop
            dup_arr (I) := dup_arr (I) + 1;
            pragma Loop_Variant(Increases => dup_arr(I)); --(I.1)

            pragma Loop_Invariant(sample(sample'Last) <= cdf_weights(cdf_weights'Last));--(I.2)

            pragma Loop_Invariant(dup_arr(I) in cdf_weights'Range);--(I.3)

            pragma Loop_Invariant(if I = sample'First then dup_arr(I) > dup_arr'Loop_Entry(I) else
     dup_arr(I) > dup_arr(I-1));--(I.4)

            pragma Loop_Invariant(sample(I) > cdf_weights(dup_arr(I)-1)
                                  and then sample(I) <= cdf_weights(dup_arr(I)-1) + big_int);--(I
    .5)
         end loop;

         pragma Loop_Invariant(for all K in dup_arr'Range => dup_arr(K) in cdf_weights'Range);--(O
    .1)

         pragma Loop_Invariant(for all K in dup_arr'First+1..I => (for all J in dup_arr'First..K-1
     => dup_arr(K) >= dup_arr(J)));--(O.2)

         pragma Loop_Invariant(if dup_arr(I) = 0 then sample(I) <= cdf_weights(dup_arr(I)) else
                                  sample(I) > cdf_weights(dup_arr(I)-1) and then sample(I) <=
    cdf_weights(dup_arr(I)));--(O.3)

         pragma Loop_Invariant(for all K in dup_arr'First..I =>
                                 (if dup_arr(K) > 0 and then K = 0 then sample(K) - cdf_weights(
    dup_arr(K)-1) in 1 .. big_int)
                                 and then (if dup_arr(K) > 0 and then K > 0 and then dup_arr(K-1)
    /= dup_arr(K)
```

```
46                                then sample(K) - cdf_weights(dup_arr(K)-1) in 1 .. big_int));--(O
   .4)
47
48          pragma Loop_Invariant(for all K in sample'First..I =>
49                                (for all J in sample'First..K =>
50                                   (if dup_arr(K) > 0 and then dup_arr(J) = dup_arr(K) then
51                                    sample(J) <= cdf_weights(dup_arr(K)) and then
52                                    sample(J) > cdf_weights(dup_arr(K)-1)) and then
53                                   (if dup_arr(K) = 0 and then dup_arr(J) = dup_arr(K) then
54                                    sample(J) <= cdf_weights(dup_arr(K)))));--(O.5)
55
56          pragma Loop_Invariant(for all K in dup_arr'First+1..I =>
57                                (for all J in dup_arr'First..K =>
58                                     (if dup_arr(K-1) = dup_arr(J) and then dup_arr(K) /= dup_arr
   (J) then
59                                           (if J = 0 or (J > 0 and then dup_arr(J) > dup_arr(J-1)
   ) then
60                                             K-J in I_Weights(dup_arr(J))/big_int ..
61                                                    I_Weights(dup_arr(J))/big_int + 1)) and then
62
63                                      (if K = sample'Last and then dup_arr(K) = dup_arr(J) then
64                                           (if J = 0 or (J > 0 and then dup_arr(J) > dup_arr(J-1)
   ) then
65                                             K+1-J in I_Weights(dup_arr(J))/big_int ..
66                                                      I_Weights(dup_arr(J))/big_int + 1))));--(O
   .6)
67
68
69       end loop;
70
71
72
73       return dup_arr;
74    end systematic_resampling;
75
76
77
78    function stratified_resampling (random_number_arr : I_Arr;
79                                    I_Weights : I_Arr)
80                                    return I_Arr
81    is
82       dup_arr : I_Arr := (others => 0);
83       cdf_weights : I_Arr := CDF(I_Weights);
84       sample : I_Arr := strat_samp_arr(random_number_arr);
85    begin
86       for I in sample'Range loop
87          if I > sample'First then
88             dup_arr (I) := dup_arr (I-1);
89          end if;
90
91          while sample(I) > cdf_weights(dup_arr(I)) loop
92             dup_arr (I) := dup_arr (I) + 1;
93             pragma Loop_Variant(Increases => dup_arr(I));--(I.1)
94
```

```
95            pragma Loop_Invariant(dup_arr(I) in cdf_weights'Range);--(I.2)
96
97            pragma Loop_Invariant(sample(sample'Last) <= cdf_weights(cdf_weights'Last));--(I.3)
98
99            pragma Loop_Invariant(if I = sample'First then dup_arr(I) > dup_arr'Loop_Entry(I) else
       dup_arr(I) > dup_arr(I-1));--(I.4)
100
101           pragma Loop_Invariant(sample(I) > cdf_weights(dup_arr(I)-1)
102                                 and then sample(I) <= cdf_weights(dup_arr(I)-1) + 2*big_int-1);
      --(I.5)
103        end loop;
104
105        pragma Loop_Invariant(for all K in dup_arr'Range => dup_arr(K) in cdf_weights'Range);--(O
      .1)
106
107        pragma Loop_Invariant(for all K in dup_arr'First+1..I => (for all J in dup_arr'First..K-1
       => dup_arr(K) >= dup_arr(J)));--(O.2)
108
109        pragma Loop_Invariant(if dup_arr(I) = 0 then sample(I) <= cdf_weights(dup_arr(I)) else
110                               sample(I) > cdf_weights(dup_arr(I)-1) and then sample(I) <=
      cdf_weights(dup_arr(I)));--(O.3)
111
112        pragma Loop_Invariant(for all J in dup_arr'First..I =>
113                              (if dup_arr(J) > 0 and then J = 0 then sample(J) - cdf_weights(
      dup_arr(J)-1) in 1 .. 2*big_int-1)
114                              and then (if dup_arr(J) > 0 and then J > 0 and then dup_arr(J-1)
      /= dup_arr(J)
115                              then sample(J) - cdf_weights(dup_arr(J)-1) in 1 .. 2*big_int-1));
      --(O.4)
116
117        pragma Loop_Invariant(for all K in sample'First..I =>
118                              (for all J in sample'First..K =>
119                                 (if dup_arr(K) > 0 and then dup_arr(J) = dup_arr(K) then
120                                  sample(J) <= cdf_weights(dup_arr(K)) and then
121                                  sample(J) > cdf_weights(dup_arr(K)-1)) and then
122                                 (if dup_arr(K) = 0 and then dup_arr(J) = dup_arr(K) then
123                                  sample(J) <= cdf_weights(dup_arr(K)))));--(O.5)
124
125        pragma Loop_Invariant(for all K in dup_arr'First+1..I =>
126                              (for all J in dup_arr'First..K =>
127                                   (if dup_arr(K-1) = dup_arr(J) and then dup_arr(K) /= dup_arr
      (J) then
128                                         (if J = 0 or (J > 0 and then dup_arr(J) > dup_arr(J-1)
      ) then
129                                           K-J in I_Weights(dup_arr(J))/big_int - 1 ..
130                                                  I_Weights(dup_arr(J))/big_int + 2)) and then
131
132                                   (if K = sample'Last and then dup_arr(K) = dup_arr(J) then
133                                         (if J = 0 or (J > 0 and then dup_arr(J) > dup_arr(J-1)
      ) then
134                                           K+1-J in I_Weights(dup_arr(J))/big_int - 1 ..
135                                                    I_Weights(dup_arr(J))/big_int + 2))));--(O
      .6)
136
```

```
137
138       end loop;
139
140       return dup_arr;
141    end stratified_resampling;
142
143
144
145
146    function multinomial_resampling(random_number_arr : I_Arr;
147                                    I_Weights : I_Arr)
148                                       return I_Arr with
149      SPARK_Mode => Off
150    is
151       dup_arr : I_Arr := (others => 0);
152       cdf_weights : I_Arr := CDF(I_Weights);
153       sample : I_Arr := random_number_arr;
154    begin
155       for I in sample'Range loop
156          if I > sample'First then
157             dup_arr (I) := dup_arr (I-1);
158          end if;
159
160          while sample(I) > cdf_weights(dup_arr(I)) loop
161             dup_arr (I) := dup_arr (I) + 1;
162          end loop;
163
164       end loop;
165
166
167
168       return dup_arr;
169    end multinomial_resampling;
170
171
172
173
174    function residual_resampling(random_number_arr : I_Arr;
175                                 Weights : F_Arr)
176                                 return I_Arr with
177      SPARK_Mode => Off
178    is
179       dup_arr : I_Arr := (others => 0);
180       min_copies : I_Arr := (others => 0);
181       residual : F_Arr := (others => 0.0);
182       I_Weights : I_Arr := (others => 0);
183       leftover : I_Arr;
184       idx : Integer := 0;
185       sum : Float := 0.0;
186    begin
187       for I in Weights'Range loop
188          min_copies(I) := Integer(Float'Truncation(Float(Weights'Length)*Weights(I)));
189       end loop;
190
```

```
191       for I in min_copies'Range loop
192          for J in 1 .. min_copies(I) loop
193             dup_arr(idx) := I;
194             idx := idx + 1;
195          end loop;
196       end loop;
197
198       for I in residual'Range loop
199          residual (I) := Weights(I) - Float(residual(I));
200          sum := sum + residual(I);
201       end loop;
202
203       for I in residual'Range loop
204          residual (I) := residual(I)/sum;
205       end loop;
206
207       I_Weights := int_weights(residual);
208       leftover := multinomial_resampling(random_number_arr, I_Weights);
209
210       while idx < dup_arr'Length loop
211          dup_arr(idx) := leftover(idx);
212          idx := idx + 1;
213          end loop;
214
215       return dup_arr;
216    end residual_resampling;
217
218
219 end resample;
```

Listing A.1: resample.adb

```
1  with helper_methods; use helper_methods;
2
3  package resample is
4     pragma SPARK_Mode;
5
6
7     function systematic_resampling (random_number : Integer; I_Weights : I_Arr) return I_Arr with
8       Pre => I_Weights'First = 0 and then I_Weights'Last+1 <= max_particles and then
9       random_number >= 1  and then random_number <= big_int and then
10      (for all J in I_Weights'Range => I_Weights(J) in 0 .. big_int*I_Weights'Length) and then
11      I_Weights (I_Weights'First) = CDF(I_Weights)(I_Weights'First) and then
12      (for all J in I_Weights'Range => (if J = 0 then I_Weights(J) = CDF(I_Weights)(J)
13                                        else I_Weights(J) = CDF(I_Weights)(J) - CDF(I_Weights)(J
      -1))),
14
15      Post => (for all K in systematic_resampling'Result'First+1..systematic_resampling'Result'Last
       =>
16                 (for all J in systematic_resampling'Result'First..K =>
17
18                       (if systematic_resampling'Result(K-1) = systematic_resampling'Result(J) and
      then
```

```
19                              systematic_resampling'Result(K) /= systematic_resampling'Result(J)
    then
20                           (if J = 0 or (J > 0 and then systematic_resampling'Result(J) >
    systematic_resampling'Result(J-1)) then
21                              K-J in I_Weights(systematic_resampling'Result(J))/big_int ..
22                                     I_Weights(systematic_resampling'Result(J))/big_int + 1)) and
    then
23
24                      (if K = systematic_resampling'Result'Last and then
25                         systematic_resampling'Result(K) = systematic_resampling'Result(J) then
26                          (if J = 0 or (J > 0 and then systematic_resampling'Result(J) >
    systematic_resampling'Result(J-1)) then
27                              K+1-J in I_Weights(systematic_resampling'Result(J))/big_int ..
28                                       I_Weights(systematic_resampling'Result(J))/big_int + 1))));
29
30    function stratified_resampling (random_number_arr : I_Arr; I_Weights : I_Arr) return I_Arr with
31      Pre => I_Weights'First = 0 and then I_Weights'Last+1 <= max_particles and then
32      (for all K in random_number_arr'Range => random_number_arr (K) in 1 .. big_int) and then
33      (for all J in I_Weights'Range => I_Weights(J) in 0 .. big_int*I_Weights'Length) and then
34      I_Weights (I_Weights'First) = CDF(I_Weights)(I_Weights'First) and then
35      (for all J in I_Weights'Range => (if J = 0 then I_Weights(J) = CDF(I_Weights)(J)
36                                           else I_Weights(J) = CDF(I_Weights)(J) - CDF(I_Weights)(J
    -1))),
37
38      Post => (for all K in stratified_resampling'Result'First+1..stratified_resampling'Result'Last
        =>
39                  (for all J in stratified_resampling'Result'First..K =>
40
41                      (if  stratified_resampling'Result(K-1) = stratified_resampling'Result(J) and
    then
42                              stratified_resampling'Result(K) /= stratified_resampling'Result(J)
    then
43                           (if J = 0 or (J > 0 and then stratified_resampling'Result(J) >
    stratified_resampling'Result(J-1)) then
44                              K-J in I_Weights(stratified_resampling'Result(J))/big_int - 1 ..
45                                     I_Weights(stratified_resampling'Result(J))/big_int + 2)) and
    then
46
47                      (if K = stratified_resampling'Result'Last and then
48                         stratified_resampling'Result(K) = stratified_resampling'Result(J) then
49                         (if J = 0 or (J > 0 and then stratified_resampling'Result(J) >
    stratified_resampling'Result(J-1)) then
50                              K+1-J in I_Weights(stratified_resampling'Result(J))/big_int - 1 ..
51                                       I_Weights(stratified_resampling'Result(J))/big_int + 2))));
52
53    function multinomial_resampling(random_number_arr : I_Arr; I_Weights : I_Arr) return I_Arr;
54
55    function residual_resampling(random_number_arr : I_Arr; Weights : F_Arr) return I_Arr;
56
57
58 end resample;
```

Listing A.2: resample.ads

Appendix A. Main Method

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Command_Line; use Ada.Command_Line;
with ada.numerics.discrete_random;
with resample; use resample;
with helper_methods; use helper_methods;


procedure Main is
   package Rand_Int is new ada.numerics.discrete_random(rand_range);
   use Rand_Int;
   G : Generator;
   random_number : Integer;
   random_number_arr : I_Arr;
   Weights : F_Arr;
   fixed_PDF : I_Arr;
   F         : File_Type;
   idx : Integer := Weights'First;
   new_idxs : I_Arr := (others => 0);
   resampling_method : String := Argument (1);
   int_weights_arr : I_Arr;

begin
   Reset (G);
   Open (F, In_File, "Input.out");
   while not End_Of_File (F) loop
      Weights (idx) := Float'Value (Get_Line (F));
      idx := idx + 1;
   end loop;
   Close (F);

   if resampling_method = "Stratified" then
      for I in random_number_arr'Range loop
         random_number_arr (I) := Random (G);
      end loop;
      int_weights_arr := int_weights(Weights);
      fixed_PDF := PDF(int_weights_arr);
      new_idxs := stratified_resampling(random_number_arr, fixed_PDF);

      Create (F, Out_File, "random.out");
      for I in new_idxs'First .. new_idxs'Last loop
         Put_Line(F, Float'Image (Float(random_number_arr(I))/Float(big_int)));
      end loop;
      Close (F);

   elsif resampling_method = "Systematic" then
      random_number := Random (G);
      int_weights_arr := int_weights(Weights);
      fixed_PDF := PDF(int_weights_arr);
      new_idxs := systematic_resampling(random_number, fixed_PDF);

      Create (F, Out_File, "random.out");
      Put_Line(F, Float'Image (Float(random_number)/Float(big_int)));
```

```
53        Close (F);
54
55     end if;
56
57     Create (F, Out_File, "Output.out");
58     for I in new_idxs'First .. new_idxs'Last loop
59       Put_Line(F, Integer'Image (new_idxs(I)));
60     end loop;
61     Close (F);
62
63  end Main;
```

Listing A.1: Main

Appendix A. Python Simulation

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.linalg as la
from math import pi, sqrt, exp
import os
import xlrd

'''
This file is divided into three parts:
1. The simulator
2. The model
3. The estimator
4. Visualization and running stuff
'''

class SatelliteSim:

    def __init__(self, Q=None, R=None, P0=None, project_loc="C:/Users/dorke/Downloads/PF_Resample/"):

        #Setup the simulation timestep.

        # Reading an excel file using Python
        # Give the location of the file
        loc = (project_loc + "pytest.xls")

        wb = xlrd.open_workbook(loc)
        sheet = wb.sheet_by_index(0)
        sheet.cell_value(0, 0)

        # init time steps
        self.num_time_stps = sheet.nrows-1

        self.time = np.zeros(sheet.nrows-1).reshape((sheet.nrows-1,1))
        self.dt = np.zeros(sheet.nrows-2).reshape((sheet.nrows-2,1))
        lat = np.zeros(sheet.nrows-1).reshape((sheet.nrows-1,1))
        lon = np.zeros(sheet.nrows-1).reshape((sheet.nrows-1,1))
        alt = np.zeros(sheet.nrows-1).reshape((sheet.nrows-1,1))
        r_lat = np.zeros(sheet.nrows-1).reshape((sheet.nrows-1,1))
        r_lon = np.zeros(sheet.nrows-1).reshape((sheet.nrows-1,1))
        r_alt = np.zeros(sheet.nrows-1).reshape((sheet.nrows-1,1))

        for i in range(sheet.nrows-1):
            self.time[i] = sheet.cell_value(i+1, 12)
            if i+2 < sheet.nrows:
                self.dt[i] = sheet.cell_value(i+2, 12) - sheet.cell_value(i+1, 12)
            lat[i] = sheet.cell_value(i+1, 13)
            lon[i] = sheet.cell_value(i+1, 14)
            alt[i] = sheet.cell_value(i+1, 15)
            r_lat[i] = sheet.cell_value(i+1, 16)
            r_lon[i] = sheet.cell_value(i+1, 17)
            r_alt[i] = sheet.cell_value(i+1, 18)
```

```
52
53
54          self.store_state = np.concatenate((lat,lon,alt),axis=1)
55          self.inputs = np.concatenate((r_lat,r_lon,r_alt),axis=1)
56
57          # Init
58          self.dt_divider = np.ceil(10.*self.dt[0])
59
60          if Q is None:
61              #.2m, 3 degrees
62              self.Q = np.eye(3)*1000000**2
63              self.Q[2,2] = 100000000
64          else:
65              self.Q = Q
66          self.S_Q = la.cholesky(self.Q*self.dt[0]/self.dt_divider)
67          if R is None:
68              self.R = np.eye(3)*10000000000000
69              self.R[2,2] = 1000000
70          else:
71              assert R.shape == (3,3)
72              self.R = R
73          self.S_R = la.cholesky(self.R)
74          if P0 is None:
75              self.P0 = np.eye(3)
76              self.P0[2,2] = .0225
77          else:
78              self.P0 = P0
79          self.S_P0 = la.cholesky(self.P0)
80
81          self.store_meas = np.zeros(self.store_state.shape)
82          for idx in range(len(self.store_state)):
83              self.store_meas[idx] = self.store_state[idx] + self.S_R.dot(np.random.randn(3))
84
85
86  class SatelliteModel:
87
88      '''
89      'initialize' will generate a set of samples from P0
90      'propagate' will only move one step forward for the x values passed in
91      'p_z_given_x' returns p(z|x) for the z and x values passed in
92      '''
93
94      def __init__(self, Q=None, R=None, P0=None, dt=None):
95          if dt is None:
96              self.dt = .1
97          else:
98              self.dt = dt
99
100         #process noise
101         if Q is None:
102             self.Q = np.eye(3)*1000000**2
103             self.Q[2,2] = 100000000
104         else:
105             self.Q = Q
```

```
106         self.S_Q = la.cholesky(self.Q)
107 
108         #measurement
109         if R is None:
110             self.R = np.eye(3)*1000000000000
111             self.R[2,2] = 100000
112         else:
113             assert R.shape == (3,3)
114             self.R = R
115         self.inv_R = la.inv(self.R)
116         self.gauss_const = 1./sqrt(la.det(2*pi*self.R))
117 
118         if P0 is None:
119             self.P0 = np.eye(3)*10000000000000
120             self.P0[2,2] = 1000000
121         else:
122             self.P0 = P0
123         self.S_P0 = la.cholesky(self.P0)
124 
125 
126     def initialize(self, init_loc,num_samples):
127         '''
128         Sample from the initial PDF.  Basically, assuming it is
129         a zero-mean Gaussian with covariance P0 (passed into __init__)
130         Args:  num_samples:  How many possible initializations you want
131         Returns: an num_samplesX3 numpy array
132         '''
133         return np.random.randn(num_samples,3).dot(self.S_P0.T) + init_loc
134 
135     def propagate(self, state, inputs, dt, with_noise=True):
136         '''
137         Runs the satellite forward with inputs V and w _for one timestep_.
138         Args:
139             state:  Can be either a (3,) numpy array or a (N,3) array
140             (Lat, Long, and Alt)
141         Returns:
142             An array the same size as state but propgated in time.
143         '''
144         my_state = state
145         if len(state.shape)==1:
146             my_state = my_state.reshape((1,3))
147         going_out = np.zeros(my_state.shape)
148 
149         for i,x in enumerate(my_state):
150             dx = np.array([inputs[0], inputs[1], inputs[2]])*dt
151             going_out[i] = x + dx
152             if with_noise:
153                 going_out[i] += self.S_Q.dot(np.random.randn(3))
154 
155         if len(state.shape)==1:
156             going_out = going_out.reshape((3,))
157 
158         return going_out
159 
```

```
160     def p_z_given_x(self, z, x):
161         '''
162         Computes the probability of receiving the measurement z given
163         the value x
164         Args:
165             z: a (2,) numpy array holding the measurement
166             x: either a (3,) or (N,3) numpy array that holds either 1 or
167                 multiple state values
168         Returns:
169             Either a scalar or a (N,) size numpy array with probabilities
170         '''
171         my_state=x
172         if len(x.shape)==1:
173             my_state = my_state.reshape((1,3))
174         going_out = np.zeros(len(my_state))
175         #Done data shaping.  Compute the actual values
176         for i,particle_x in enumerate(my_state):
177             exponent = -0.5 * (z-particle_x).dot(self.inv_R.dot(z-particle_x))
178             going_out[i] = self.gauss_const * exp(exponent)
179 
180         if len(x.shape)==1:
181             going_out = going_out[0].item()
182 
183         return going_out
184 
185 class ParticleFilter:
186     def __init__(self,model, init_loc ,N=500,resample_threshold=200, method=None):
187         self.particles = model.initialize(init_loc, N)
188         self.weights = np.ones(N)/float(N)
189         self.N = N
190         self.resample_threshold = resample_threshold
191         #Used for systematic resampling
192         self.steps = np.ones(N)/float(N)
193         self.steps[0] = 0
194         self.steps = np.cumsum(self.steps)
195         self.resampling_method = method
196         self.model = model
197 
198     def compute_Neff(self):
199         return 1/np.sum(np.square(self.weights))
200 
201     def resample(self):
202         # python code for systematic resampling
203         if self.resampling_method == "Systematic":
204             random_num = np.loadtxt("random.out",dtype= float, unpack=False)
205             cdf_weights = np.cumsum(self.weights)
206             sampling_arr = random_num/float(self.N) + self.steps
207             py_idxs = np.searchsorted(cdf_weights, sampling_arr)
208 
209         elif self.resampling_method == "Stratified":
210             random_arr = np.loadtxt("random.out",dtype= float, unpack=False)
211             cdf_weights = np.cumsum(self.weights)
212             sampling_arr = random_arr/float(self.N) + self.steps
213             py_idxs = np.searchsorted(cdf_weights, sampling_arr)
```

```
214
215         # Ada
216         ada_idxs = np.loadtxt("Output.out",dtype= int, unpack=False)
217         # Check Ada vs Python
218         check = np.concatenate((py_idxs.reshape(500,1), ada_idxs.reshape(500,1)), axis=1)
219         np.savetxt("check.out", check,  fmt='%i, %i', header="Python, Ada")
220         # Update particles and reset weights
221         self.particles = self.particles[ada_idxs]
222         self.weights = np.ones(self.N)/float(self.N)
223
224
225
226     def step(self, meas, inputs,dt, project_loc):
227         #Check if we need to resample and resample
228         if self.compute_Neff() < self.resample_threshold:
229             np.savetxt("Input.out", pf.weights,  fmt='%.18f')
230             os.system(project_loc + "obj/main " + self.resampling_method)
231             self.resample()
232             open('Output.out', 'w').close()
233             open('Input.out', 'w').close()
234         #propagate forward if needed
235         self.particles = self.model.propagate(self.particles,inputs,dt)
236         #Apply the measurement
237         self.weights *= self.model.p_z_given_x(meas,self.particles)
238         self.weights /= np.sum(self.weights)
239
240     def get_average(self):
241         return np.average(self.particles,axis=0,weights=self.weights)
242
243 def plot_states(states, meas=None):
244     fig = plt.figure()
245     ax = fig.add_subplot(projection='3d')
246     ax.plot(states[:,0], states[:,1], states[:,2], label='Satellite', c='b')
247     if meas is not None:
248         ax.scatter(meas[:,0], meas[:,1], meas[:,2],marker='.', label='Measurements', c='r')
249     ax.set_xlabel('Lat Label')
250     ax.set_ylabel('Lon Label')
251     ax.set_zlabel('Alt Label')
252     plt.show()
253
254 def weight_to_size(weights, max_size=100, min_size=1):
255     '''
256     This function attempts to take a set of weights from the particles
257     for a particle filter and turn them into a set of sizes.  This allows
258     matplotlib's scatter to plot the particles with different sizes
259     Args:
260         weights:  The weights on the samples to be graphed
261         max_size: The size of the biggest dot that will be graphed
262         min_size: The size of the smallest dot that will be graphed
263     Returns:
264         A numpy array of the same size as weights with sizes that can be passed
265         in as the "size" parameter into plt.scatter
266     '''
267     # Let's try a linear scaling from weights to size...
```

```
268     max_w = np.max(weights)
269     min_w = np.min(weights)
270     if max_w==min_w:
271         return np.ones(len(weights))*10
272     else:
273         return np.rint(weights * (max_size-min_size)/(max_w-min_w) + min_size).astype(int)
274 
275 def plot_particles(ax, particles, weights, truth_hist, time):
276     ax.plot(truth_hist[:,0], truth_hist[:,1], truth_hist[:,2], label='Satellite')
277     sizes = weight_to_size(weights)
278     ax.scatter(particles[:,0], particles[:,1], particles[:,2], marker='o', label='Measurements', c
        ='g',s=sizes)
279     plt.title(f'time is {time}seconds')
280     plt.show()
281 
282 if __name__ == "__main__":
283     loc = "C:/Users/dorke/Downloads/PF_Resample/"
284     methods = ["Stratified", "Systematic"]
285     Resample_Method = methods[0]
286 
287     us = SatelliteSim(project_loc = loc)
288     plot_states(us.store_state, us.store_meas)
289     #Now let's run the estimator
290     um = SatelliteModel(dt=us.dt)
291     pf= ParticleFilter(um,us.store_meas[0,:], N=500,method=Resample_Method)
292     #Now run the particle filter
293     for i in range(us.num_time_stps-1):
294         if i%10 == 0:
295             #plot the state...
296             fig = plt.figure()
297             ax = fig.add_subplot(projection='3d')
298             plot_particles(ax, pf.particles, pf.weights, us.store_state[:i+1], us.time[i])
299             #plt.plot(pf.weights)
300             #plt.show()
301         pf.step(us.store_meas[i+1], us.inputs[i], us.dt[i], loc)
```

Listing A.1: Particle Filter Simulation

Bibliography

1. A. Pervan and T. Murphey, Algorithmic Materials: Embedding Computation within Material Properties for Autonomy. Mechanical Engineering, Northwestern University, Evanston IL 60208.

2. A. Hayes, "How does systematic sampling work?," Investopedia, 10-Jan-2022. [Online]. Available: https://www.investopedia.com/terms/s/systematic-sampling.asp. [Accessed: 10-Jan-2022].

3. "'learn.adacore.com,'" learn.adacore.com. [Online]. Available: https://learn.adacore.com/courses/intro-to-spark/index.html. [Accessed: 10-Jan-2022].

4. "5.6. assertion pragmas," 5.6. Assertion Pragmas - SPARK User's Guide 23.0w. [Online]. Available: https://docs.adacore.com/spark2014-docs/html/ug/en/source/assertion_pragmas.html#loop-invariants. [Accessed: 10-Jan-2022].

5. J. W. McCormick and P. C. Chapin, "Building high integrity applications with Spark," UNI ScholarWorks. [Online]. Available: https://scholarworks.uni.edu/facbook/118/. [Accessed: 10-Jan-2022].

6. C. Dross and Y. Moy, Introduction to SPARK, 2020. AdaCore, 2021.

7. R. Amiard and G. A. Hoffmann, Introducation to Ada. 2021.

8. B. J. G. Presslie, Programming in ADA 2012. Cambridge: Cambridge university press, 2014.

9. "What is numpy?," What is NumPy? - NumPy v1.21 Manual. [Online]. Available: https://numpy.org/doc/1.21/user/whatisnumpy.html. [Accessed: 10-Jan-2022].

10. “Math - mathematical functions,” math - Mathematical functions - Python 3.10.1 documentation. [Online]. Available: https://docs.python.org/3/library/math.html. [Accessed: 10-Jan-2022].

11. “Matplotlib.pyplot,” matplotlib.pyplot - Matplotlib 3.5.1 documentation. [Online]. Available: https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.html. [Accessed: 10-Jan-2022].

12. “OS - miscellaneous operating system interfaces,” os - Miscellaneous operating system interfaces - Python 3.10.1 documentation. [Online]. Available: https://docs.python.org/3/library/os.html. [Accessed: 10-Jan-2022].

13. “XLRD,” xlrd. [Online]. Available: https://xlrd.readthedocs.io/en/latest/. [Accessed: 10-Jan-2022].

14. “What’s new in python 3.10,” What’s New In Python 3.10 - Python 3.10.1 documentation. [Online]. Available: https://docs.python.org/3/whatsnew/3.10.html. [Accessed: 10-Jan-2022].

15. Milford, Michael J., and Gordon F. Wyeth. “SeqSLAM: Visual Route-Based Navigation for Sunny Summer Days and Stormy Winter Nights.” Proceedings - IEEE International Conference on Robotics and Automation. Institute of Electrical and Electronics Engineers Inc., 2012. 1643–1649. Proceedings - IEEE International Conference on Robotics and Automation. Web.

16. X. Wang, T. Li, S. Sun, and J. Corchado, “A survey of recent advances in particle filters and remaining challenges for multitarget tracking,” Sensors, vol. 17, no. 12, p. 2707, 2017.

17. K. Manandhar, X. Cao, F. Hu and Y. Liu, "Detection of Faults and Attacks Including False Data Injection Attack in Smart Grid Using Kalman Filter," in IEEE Transactions on Control of Network Systems, vol. 1, no. 4, pp. 370-379, Dec. 2014, doi: 10.1109/TCNS.2014.2357531.

18. N. Hirose, S. Taguchi, F. Xia, R. Martin-Martin, Y. Tahara, M. Ishigaki, and S. Savarese, "Probabilistic Visual Navigation with Bidirectional Image Prediction," arXiv.org, 20-Mar-2020. [Online]. Available: https://arxiv.org/abs/2003.09224. [Accessed: 21-Feb-2021].

19. M. Sabatini, G. B. Palmerini, and P. Gasbarri, "A testbed for visual based navigation and control during space rendezvous operations," Acta Astronautica, 03-Aug-2015. [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/S0094576515003070?via

20. Y. Qiu, A. Pal, and H. I. Christensen, "Learning hierarchical relationships for object-goal navigation," arXiv.org, 18-Nov-2020. [Online]. Available: https://arxiv.org/abs/2003.06749. [Accessed: 21-Feb-2021].

21. M. Sabatini, R. Monti, P. Gasbarri, and G. B. Palmerini, "Adaptive and robust algorithms and tests for visual-based navigation of a space robotic manipulator," Acta Astronautica, 19-Nov-2012. [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/S0094576512003943?via

22. S. Ishikawa, H. Kuwamoto and S. Ozawa, "Visual navigation of an autonomous vehicle using white line recognition," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 10, no. 5, pp. 743-749, Sept. 1988, doi: 10.1109/34.6786.

23. O. Kupervasser, H. Kutomanov, O. Levi, V. Pukshansky, and R. Yavich, "Using Deep Learning for Visual Navigation of Drone with Respect to 3D Ground Objects," Mathematics, vol. 8, no. 12, p. 2140, 2020.

24. G. Bianchi and I. Tinnirello, "Kalman filter estimation of the number of competing terminals in an IEEE 802.11 network," IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No.03CH37428), San Francisco, CA, USA, 2003, pp. 844-852 vol.2, doi: 10.1109/INFCOM.2003.1208922.

25. Y. Iiguni, H. Sakai and H. Tokumaru, "A real-time learning algorithm for a multilayered neural network based on the extended Kalman filter," in IEEE Transactions on Signal Processing, vol. 40, no. 4, pp. 959-966, April 1992, doi: 10.1109/78.127966.

26. L. Guo, "Estimating time-varying parameters by the Kalman filter based algorithm: stability and convergence," in IEEE Transactions on Automatic Control, vol. 35, no. 2, pp. 141-147, Feb. 1990, doi: 10.1109/9.45169.

27. G. V. Puskorius and L. A. Feldkamp, "Decoupled extended Kalman filter training of feedforward layered networks," IJCNN-91-Seattle International Joint Conference on Neural Networks, Seattle, WA, USA, 1991, pp. 771-777 vol.1, doi: 10.1109/IJCNN.1991.155276.

28. R. J. Williams, "Training recurrent networks using the extended Kalman filter," [Proceedings 1992] IJCNN International Joint Conference on Neural Networks, Baltimore, MD, USA, 1992, pp. 241-246 vol.4, doi: 10.1109/IJCNN.1992.227335.

29. S. Sangsuk-Iam and T. E. Bullock, "Analysis of discrete-time Kalman filtering under incorrect noise covariances," in IEEE Transactions on Automatic Control, vol. 35, no. 12, pp. 1304-1309, Dec. 1990, doi: 10.1109/9.61006.

30. R. Van der Merwe and E. A. Wan, "The square-root unscented Kalman filter for state and parameter-estimation," 2001 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings (Cat. No.01CH37221), Salt Lake City, UT, USA, 2001, pp. 3461-3464 vol.6, doi: 10.1109/ICASSP.2001.940586.

31. Y. Song and J. W. Grizzle, "The Extended Kalman Filter as a Local Asymptotic Observer for Nonlinear Discrete-Time Systems," 1992 American Control Conference, Chicago, IL, USA, 1992, pp. 3365-3369, doi: 10.23919/ACC.1992.4792775.

32. Xiangheng Liu and A. Goldsmith, "Kalman filtering with partial observation losses," 2004 43rd IEEE Conference on Decision and Control (CDC) (IEEE Cat. No.04CH37601), Nassau, Bahamas, 2004, pp. 4180-4186 Vol.4, doi: 10.1109/CDC.2004.1429408.

33. K. K. C. Yu, N. R. Watson and J. Arrillaga, "An adaptive Kalman filter for dynamic harmonic state estimation and harmonic injection tracking," in IEEE Transactions on Power Delivery, vol. 20, no. 2, pp. 1577-1584, April 2005, doi: 10.1109/TPWRD.2004.838643.

34. M. Boutayeb, H. Rafaralahy and M. Darouach, "Convergence analysis of the extended Kalman filter used as an observer for nonlinear deterministic discrete-time systems," in IEEE Transactions on Automatic Control, vol. 42, no. 4, pp. 581-586, April 1997, doi: 10.1109/9.566674.

35. S. Bolognani, L. Tubiana and M. Zigliotto, "Extended Kalman filter tuning in sensorless PMSM drives," in IEEE Transactions on Industry Applications, vol. 39, no. 6, pp. 1741-1747, Nov.-Dec. 2003, doi: 10.1109/TIA.2003.818991.

36. Chien-Shu Hsieh, "Robust two-stage Kalman filters for systems with unknown inputs," in IEEE Transactions on Automatic Control, vol. 45, no. 12, pp. 2374-2378, Dec. 2000, doi: 10.1109/9.895577.

37. X. Yun and E. R. Bachmann, "Design, Implementation, and Experimental Results of a Quaternion-Based Kalman Filter for Human Body Motion Tracking," in IEEE Transactions on Robotics, vol. 22, no. 6, pp. 1216-1227, Dec. 2006, doi: 10.1109/TRO.2006.886270.

38. Taek Song and J. Speyer, "A stochastic analysis of a modified gain extended Kalman filter with applications to estimation with bearings only measurements," in IEEE Transactions on Automatic Control, vol. 30, no. 10, pp. 940-949, October 1985, doi: 10.1109/TAC.1985.1103821.

39. L. Jetto, S. Longhi and G. Venturini, "Development and experimental validation of an adaptive extended Kalman filter for the localization of mobile robots," in IEEE Transactions on Robotics and Automation, vol. 15, no. 2, pp. 219-229, April 1999, doi: 10.1109/70.760343.

———-

40. Haili Ma and A. A. Girgis, "Identification and tracking of harmonic sources in a power system using a Kalman filter," in IEEE Transactions on Power Delivery, vol. 11, no. 3, pp. 1659-1665, July 1996, doi: 10.1109/61.517531.

41. P. K. Dash, R. K. Jena, G. Panda and A. Routray, "An extended complex Kalman filter for frequency measurement of distorted signals," in IEEE Transactions on

Instrumentation and Measurement, vol. 49, no. 4, pp. 746-753, Aug. 2000, doi: 10.1109/19.863918.

42. J. L. Marins, Xiaoping Yun, E. R. Bachmann, R. B. McGhee and M. J. Zyda, "An extended Kalman filter for quaternion-based orientation estimation using MARG sensors," Proceedings 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems. Expanding the Societal Role of Robotics in the the Next Millennium (Cat. No.01CH37180), Maui, HI, USA, 2001, pp. 2003-2011 vol.4, doi: 10.1109/IROS.2001.976367.

43. L. Hostetler and R. Andreas, "Nonlinear Kalman filtering techniques for terrain-aided navigation," in IEEE Transactions on Automatic Control, vol. 28, no. 3, pp. 315-323, March 1983, doi: 10.1109/TAC.1983.1103232.

44. S. J. Julier and J. J. LaViola, "On Kalman Filtering With Nonlinear Equality Constraints," in IEEE Transactions on Signal Processing, vol. 55, no. 6, pp. 2774-2784, June 2007, doi: 10.1109/TSP.2007.893949.

45. S. Sarkka and A. Nummenmaa, "Recursive Noise Adaptive Kalman Filtering by Variational Bayesian Approximations," in IEEE Transactions on Automatic Control, vol. 54, no. 3, pp. 596-600, March 2009, doi: 10.1109/TAC.2008.2008348.

46. M. St-Pierre and D. Gingras, "Comparison between the unscented Kalman filter and the extended Kalman filter for the position estimation module of an integrated navigation information system," IEEE Intelligent Vehicles Symposium, 2004, Parma, Italy, 2004, pp. 831-835, doi: 10.1109/IVS.2004.1336492.

47. H. M. Beides and G. T. Heydt, "Dynamic state estimation of power system harmonics using Kalman filter methodology," in IEEE Transactions on Power Delivery, vol. 6, no. 4, pp. 1663-1670, Oct. 1991, doi: 10.1109/61.97705.

48. M. S. Grewal, V. D. Henderson and R. S. Miyasako, "Application of Kalman filtering to the calibration and alignment of inertial navigation systems," 29th IEEE Conference on Decision and Control, Honolulu, HI, USA, 1990, pp. 3325-3334 vol.6, doi: 10.1109/CDC.1990.203410.

49. S. Rezaei and R. Sengupta, "Kalman Filter-Based Integration of DGPS and Vehicle Sensors for Localization," in IEEE Transactions on Control Systems Technology, vol. 15, no. 6, pp. 1080-1088, Nov. 2007, doi: 10.1109/TCST.2006.886439.

50. J. L. Crassidis, "Sigma-point Kalman filtering for integrated GPS and inertial navigation," in IEEE Transactions on Aerospace and Electronic Systems, vol. 42, no. 2, pp. 750-756, April 2006, doi: 10.1109/TAES.2006.1642588.

51. J. J. LaViola, "A comparison of unscented and extended Kalman filtering for estimating quaternion motion," Proceedings of the 2003 American Control Conference, 2003., Denver, CO, USA, 2003, pp. 2435-2440 vol.3, doi: 10.1109/ACC.2003.1243440.

52. Y. Li, Z. Li and L. Chen, "Dynamic State Estimation of Generators Under Cyber Attacks," in IEEE Access, vol. 7, pp. 125253-125267, 2019, doi: 10.1109/ACCESS.2019.2939055.

53. G. Reina, A. Vargas, K. Nagatani and K. Yoshida, "Adaptive Kalman Filtering for GPS-based Mobile Robot Localization," 2007 IEEE International Workshop on Safety, Security and Rescue Robotics, Rome, Italy, 2007, pp. 1-6, doi: 10.1109/SSRR.2007.4381270.

54. J. Wang, C. Shyi, T. -. Hou and C. P. Fong, "Design and implementation of augmented reality system collaborating with QR code," 2010 International Computer

Symposium (ICS2010), Tainan, Taiwan, 2010, pp. 414-418, doi: 10.1109/COMPSYM.2010.5685477.

55. M. Becker, E. Regnath, and S. Chakraborty, "Development and Verification of a Flight Stack for a High-Altitude Glider in Ada/SPARK 2014," Lecture Notes in Computer Science, pp. 105–116, 2017.

56. U. Herberth, J. Rende and H. Lutz, "Development of Inertial Sensors for AHRS considering DO-254," 2018 DGON Inertial Sensors and Systems (ISS), Braunschweig, Germany, 2018, pp. 1-19, doi: 10.1109/InertialSensors.2018.8577187.

57. S. Tan, D. De, W. Song, J. Yang and S. K. Das, "Survey of Security Advances in Smart Grid: A Data Driven Approach," in IEEE Communications Surveys & Tutorials, vol. 19, no. 1, pp. 397-422, First Quarter 2017, doi: 10.1109/COMST.2016.2616442.

58. Y. J. Kim, Y. K. Chung and B. G. Lee, "Vessel tracking vision system using a combination of Kaiman filter, Bayesian classification, and adaptive tracking algorithm," 16th International Conference on Advanced Communication Technology, Pyeongchang, Korea (South), 2014, pp. 196-201, doi: 10.1109/ICACT.2014.6778948.

59. W. Gao, W. Liang and K. K. Tan, "Kalman filter based vision-servoing alignment system for human ear surgery," 2015 World Congress on Industrial Control Systems Security (WCICSS), London, UK, 2015, pp. 83-88, doi: 10.1109/WCICSS.2015.7420331.

60. B. F. La Scala and R. R. Bitmead, "Design of an extended Kalman filter frequency tracker," in IEEE Transactions on Signal Processing, vol. 44, no. 3, pp. 739-742, March 1996, doi: 10.1109/78.489052.

61. Fuwen Yang, Zidong Wang and Y. S. Hung, "Robust Kalman filtering for discrete time-varying uncertain systems with multiplicative noises," in IEEE Transactions on Automatic Control, vol. 47, no. 7, pp. 1179-1183, July 2002, doi: 10.1109/TAC.2002.800668.

62. H. G. de Marina, F. J. Pereda, J. M. Giron-Sierra and F. Espinosa, "UAV Attitude Estimation Using Unscented Kalman Filter and TRIAD," in IEEE Transactions on Industrial Electronics, vol. 59, no. 11, pp. 4465-4474, Nov. 2012, doi: 10.1109/TIE.2011.2163913.

63. E. Foxlin, "Inertial head-tracker sensor fusion by a complementary separate-bias Kalman filter," Proceedings of the IEEE 1996 Virtual Reality Annual International Symposium, Santa Clara, CA, USA, 1996, pp. 185-194, doi: 10.1109/VRAIS.1996.490527.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
03/25/2022	Master's Thesis	Sept 2020 --- Mar 2022

4. TITLE AND SUBTITLE
Formal SPARK Verification of Various Resampling Methods in Particle Filters

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Osiris J. Terry

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT-ENG-MS-22-M-067

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFRL/RQQA
Larua Humphry
WPAFB OH 45433-7765
COMM (937) 713-7032
Email: laura.humphrey@us.af.mil

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RQQA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The software verification in this thesis concentrates on verifying a particle filter for use in tracking and estimation, a key application area for the Air Force. The development and verification process described in this thesis is a demonstration of the power, limitation, and compro- mises involved in applying automated software verification tools to critical embedded software applications.

15. SUBJECT TERMS
Particle Filters, Resampling Methods, Software Verification, SPARK

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dr. Kenneth Hopkinson, AFIT/ENG
					19b. TELEPHONE NUMBER *(Include area code)*
U	U	U	UU	107	(937) 255-3636

INSTRUCTIONS FOR COMPLETING SF 298

1. REPORT DATE. Full publication date, including day, month, if available. Must cite at least the year and be Year 2000 compliant, e.g. 30-06-1998; xx-06-1998; xx-xx-1998.

2. REPORT TYPE. State the type of report, such as final, technical, interim, memorandum, master's thesis, progress, quarterly, research, special, group study, etc.

3. DATE COVERED. Indicate the time during which the work was performed and the report was written, e.g., Jun 1997 - Jun 1998; 1-10 Jun 1996; May - Nov 1998; Nov 1998.

4. TITLE. Enter title and subtitle with volume number and part number, if applicable. On classified documents, enter the title classification in parentheses.

5a. CONTRACT NUMBER. Enter all contract numbers as they appear in the report, e.g. F33315-86-C-5169.

5b. GRANT NUMBER. Enter all grant numbers as they appear in the report. e.g. AFOSR-82-1234.

5c. PROGRAM ELEMENT NUMBER. Enter all program element numbers as they appear in the report, e.g. 61101A.

5e. TASK NUMBER. Enter all task numbers as they appear in the report, e.g. 05; RF0330201; T4112.

5f. WORK UNIT NUMBER. Enter all work unit numbers as they appear in the report, e.g. 001; AFAPL30480105.

6. AUTHOR(S). Enter name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. The form of entry is the last name, first name, middle initial, and additional qualifiers separated by commas, e.g. Smith, Richard, J, Jr.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES). Self-explanatory.

8. PERFORMING ORGANIZATION REPORT NUMBER. Enter all unique alphanumeric report numbers assigned by the performing organization, e.g. BRL-1234; AFWL-TR-85-4017-Vol-21-PT-2.

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES). Enter the name and address of the organization(s) financially responsible for and monitoring the work.

10. SPONSOR/MONITOR'S ACRONYM(S). Enter, if available, e.g. BRL, ARDEC, NADC.

11. SPONSOR/MONITOR'S REPORT NUMBER(S). Enter report number as assigned by the sponsoring/ monitoring agency, if available, e.g. BRL-TR-829; -215.

12. DISTRIBUTION/AVAILABILITY STATEMENT. Use agency-mandated availability statements to indicate the public availability or distribution limitations of the report. If additional limitations/ restrictions or special markings are indicated, follow agency authorization procedures, e.g. RD/FRD, PROPIN, ITAR, etc. Include copyright information.

13. SUPPLEMENTARY NOTES. Enter information not included elsewhere such as: prepared in cooperation with; translation of; report supersedes; old edition number, etc.

14. ABSTRACT. A brief (approximately 200 words) factual summary of the most significant information.

15. SUBJECT TERMS. Key words or phrases identifying major concepts in the report.

16. SECURITY CLASSIFICATION. Enter security classification in accordance with security classification regulations, e.g. U, C, S, etc. If this form contains classified information, stamp classification level on the top and bottom of this page.

17. LIMITATION OF ABSTRACT. This block must be completed to assign a distribution limitation to the abstract. Enter UU (Unclassified Unlimited) or SAR (Same as Report). An entry in this block is necessary if the abstract is to be limited.