

Parallelizing a Particle Filter Implementation for a Two-Dimensional Point Vortex Fluid Model

Adam Mallen, Erin Soderberg

adam.mallen@marquette.edu, erin.soderberg@marquette.edu

Department of Mathematics, Statistics, and Computer Science
Marquette University
P.O. Box 1881, Milwaukee, WI 53201

Contact Author: Adam Mallen

Email: adam.mallen@marquette.edu

Phone: (815) 979-6710

Parallelizing a Particle Filter Implementation for a Two-Dimensional Point Vortex Fluid Model

ABSTRACT

Realistic models of fluid dynamics in the ocean are a useful tool for oceanographers. For instance they can help understand how oil leaked in an oil spill will spread across the surface of the ocean. One tool for creating such models is the data collected by tracers dropped in the ocean. By using the data collected by the movement of such tracers, researchers can improve their model predictions of ocean flow. However, incorporating this data into flow models is very computationally expensive. This is due to the Lagrangian nature of these measurements, which, when incorporated into a system of differential equations, introduces computational complexity. It is also due to the non-deterministic nature of the stochastic differential equations which govern the dynamics frequently used to model ocean flows. In this project, we implement a particle filtering data assimilation scheme on a two-dimensional point-vortex fluid model in parallel on the Pere cluster in order to improve the computational efficiency and speed of the particle filter.

Keywords: Data Assimilation, Point Vortex Model, Particle Filter, Kalman Filter, Parallel Programming, Condor.

1. INTRODUCTION

Modeling ocean dynamics often involves some form of data assimilation to estimate unmeasurable components of the system model. Kalman filters and particle filters have been used as data assimilation techniques for such models (Kuznetsov et al., 2003),(Spiller et al., 2008). Kalman filters are useful when the system is linear; however, particle filters are more generally applicable since they make no requirements on the system. This work focuses on a particle filter implementation of the point vortex model of fluid flows, which models these flows as a system of stochastic differential equations. A classic problem with using stochastic differential equations for modeling any system is the large amount of computational resources necessary for the numerous simulations required to generate meaningful results. Because of the randomness inherent in the dynamics of the model, each realization of such a system will be different--sometimes dramatically different. So, to get a statistical sense of the performance and error of a data assimilation scheme, we must implement the assimilation technique on a large number of realizations of the system.

The goal of this project is to take the particle filter implemented in Spiller et al, and rework it to run in parallel on the Pere cluster using the Condor job scheduler. The hundreds of simulations required to generate

meaningful performance metrics can be run in parallel, since each realization of the system is independent. Executing multiple simulations in parallel is desirable because it will greatly reduce the computation time, allowing researchers more time to analyze their results and debug their code.

2. BACKGROUND

In this section, we describe in detail the major concepts underlying the work presented in this paper.

2.1 Point Vortex Model

The point-vortex model of fluid flows is a simplified version of realistic fluid processes found in the ocean. Our model consists of two vortices of equal strengths—which rotate around each other—and a single tracer—which floats around the two vortices according to the fluid flow generated by these vortices. We know the differential equations which describe the flow depicted in Figure 1.

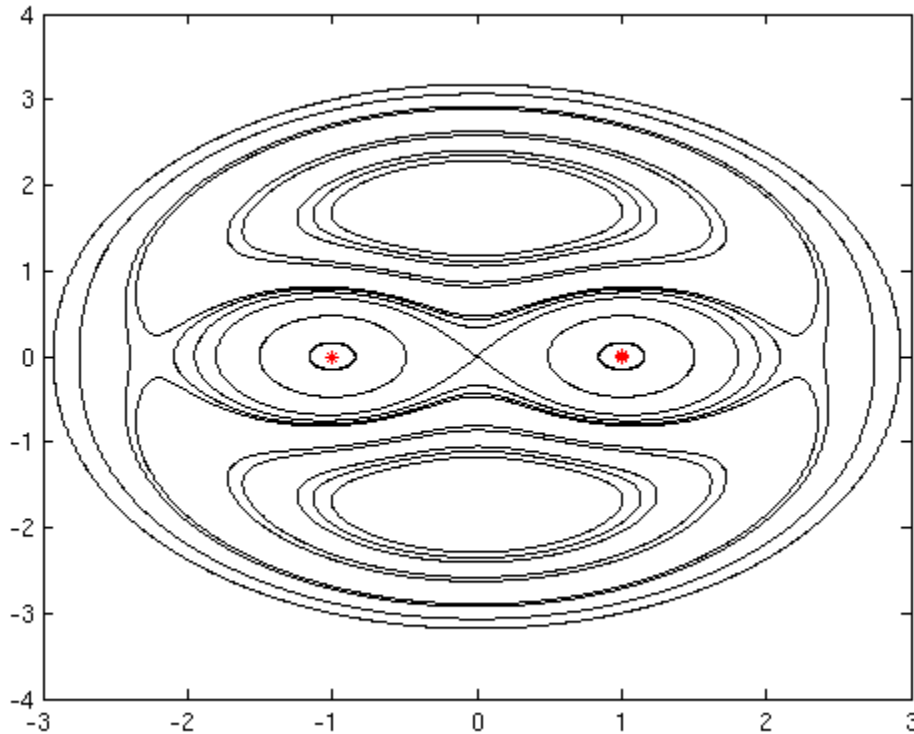


Figure 1: Point Vortex Model.

The red points represent the vortices and each black curve represents a single realization of the flow of a tracer. This picture shows the path of the tracer after rotating the frame relative to the rotation of the vortices. Also, this figure represents sample paths of the tracer with no noise component in the system.

The dynamics which drive the motion of the tracer and vortices contains a stochastic component to account for the many complex factors which affect the flow in reality, but are not taken into account in our simplified model. Thus,

we model the dynamics of our system as a stochastic differential equation to account for the noise inherent in the system.

2.2 Eulerian vs. Lagrangian Frame of Reference

Many models in oceanography use Eulerian variables, that is, variables computed on a fixed grid in space. For instance, a buoy at a fixed location in the water could provide water velocity data from its one specific location. However, much of the data available to oceanographers comes from ocean drifters and floats, examples of Lagrangian meters, which are distributed non-uniformly over the space, and do not provide data in terms of the Eulerian model variables. The data generated by these Lagrangian meters is from a frame of reference that is moving with respect to the fixed Eulerian system. Utilizing the Lagrangian data to describe the behavior of the underlying system is an important but computationally expensive task.

2.3 Data Assimilation

Data assimilation is the process of taking noisy partial observations of a system and using these observations to make statistical inferences about the parameters of the model driving the underlying dynamics. In our project, we make observations (also with measurement noise) of only the position of the tracer (a Lagrangian meter), while the position and strengths of the vortices (Eulerian variables) remain unknown. We use a data assimilation scheme for estimating the position and strengths of the vortices based on the statistical likelihood of those vortex parameters yielding the given tracer position observations.

2.4 Particle Filtering

Filtering is a common data assimilation technique used in these kinds of problems. The objective of any filtering technique is to estimate a system's hidden states (hidden parameters) by using noisy partial observations of the system to help inform our model's predictions. To do so, we can think of a probability density of each state variable at each time step and use these to make estimates on the true value of the system's state. The densities of the state variables evolve through time according to the system model and at each observation are updated according to the likelihood of the current observation. If the model is linear and the observation error is normally distributed, then a direct application of Bayes' rule to the process above yields the well known Kalman filter. Kalman filter techniques work well on high-dimensional problems but require a linear model. Alternatively, we can approximate these densities by using particle filters which make no assumptions on the linearity of the model or distribution of the

observation errors. Particle filters use a large number of random samples as a discrete approximation of these densities. Often particle filters will implement some kind of resampling scheme to sort out samples which stray too far from the observations. A resampling scheme selects only a subset of the random samples (called “particles”) at each observation time to further evolve to the next observation time. The specific subset is chosen to be the particles with the highest likelihood of yielding the current observation. Particle filters work well on nonlinear problems, but become computationally unwieldy as the state space dimension increases because a very large cloud of particles is necessary for a reasonable sampling. Because of this, simulation runs can take a long time and it would be advantageous to run the many trials required in parallel to decrease the computation time.

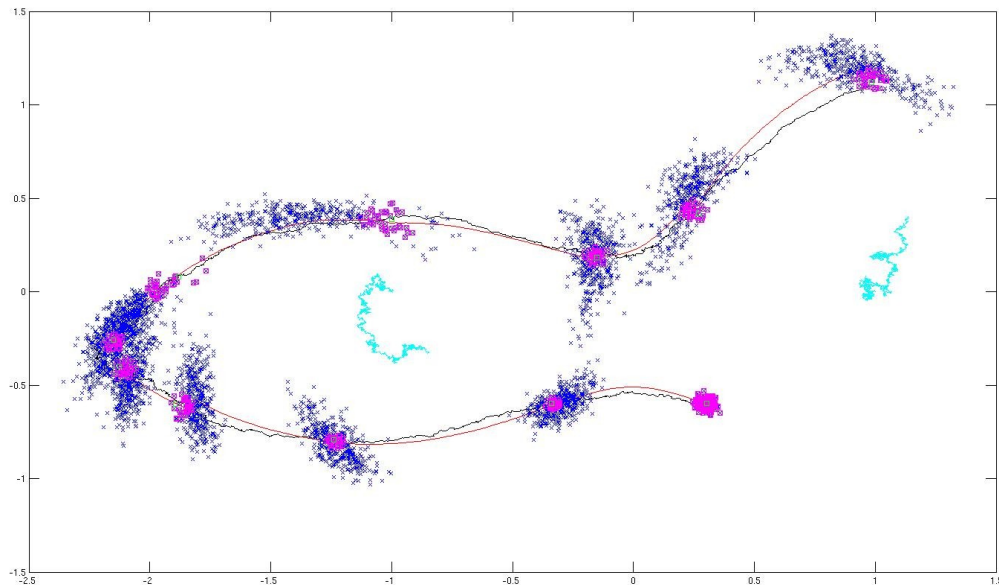


Figure 2: Illustration of particle filter implemented on tracer data.

The blue and pink marks represent particles, and the pink marks represent those particles which were kept after resampling. The cyan marks represent the true locations of the vortices. The green boxes represent the tracer location observations. The black and red paths represent the true and predicted tracer movement, respectively.

3. RELATED WORK

In “A Method for Assimilation of Lagrangian Data”, the authors present a method for incorporating Lagrangian tracer positions into a model of flows (such as ocean currents) (Kuznetsov et al., 2003). Such models are important in understanding the behavior of atmospheric and oceanic systems. Previous models relied only on Eulerian variables computed on a fixed grid in space and neglected the tracer positions because they are distributed non-

uniformly and do not return data in terms of the Eulerian variables. Their research concludes that by including Lagrangian data collected from one or two tracers, using an extended Kalman Filter, they were able to track both the tracers and the vortices. However, one problem with their work is that the method they used for forecasting the error covariance matrix is computationally expensive, and their system only works properly when the modeled noise levels are small enough. The performance of their model also depends on the initial positions of the tracer and the dynamics of the underlying system.

In “Modified particle filter methods for assimilating Lagrangian data into a point-vortex model,” the authors expand upon the work described above by using various particle filter schemes and compare their methods with the performance of the extended Kalman filter (Spiller et al., 2008). To address some of the performance issues from the earlier work, the authors introduce backtracking to account for when the predicted tracer locations have great deviation from the observed locations. Specifically, they implement cloud expanding and directed doubling backtrack particle filters, both of which achieved very low failure rates, especially when compared to the extended Kalman filter. One particular advantage of the new methods is that they still perform well even when observations are taken less frequently. However, these methods can still become very computationally expensive and lengthy.

For our project, we implement the method described in the second paper in parallel on the Pere cluster, thus decreasing the amount of time it takes to run one simulation.

4. CONCEPT DESIGN

The goal of this project is to run the multiple trials that make up a single simulation experiment in parallel. Notice on the following task dependency graphs (figures 3, 4 and 5), that there are three different points where parallelization is possible: across experiments, across trials, and across particles. Parallelizing across experiments is trivial and had already been successfully implemented prior to this work. However, since the cloud of particles in the particle filter is represented as a matrix, all the operations performed on a single particle can be translated to matrix operations performed at once on the entire cloud. Because Matlab is optimized for matrix operations, we believe that little speed up would be achieved by parallelizing across particles.

4.1 Definitions

The following definitions describe the various tasks that go into the particle filter simulation. Figures 3, 4 and 5 later in this section show the task dependency graphs for each of the components of the simulation program.

Program Run:

A program run is defined to be M experiment runs with different parameter files. The parameter files contain model and program parameters as well as initial conditions for the differential equations which govern the model. Before running the experiments the program must set up directories for the input and output files. Similarly, after running the experiments the program must clean up and organize files and output.

Experiment:

An experiment loads in all system parameters and initial conditions from a parameter file and then runs N trials with the given parameters and initial conditions. Since each experiment uses the same parameters and initial conditions, the differences between the individual trials that make up an experiment are due entirely to the stochastic components of the model. All the resulting error statistics are saved to output files.

Trial:

A trial first generates one 'truth' realization of the system and then runs one full implementation of the particle filter with generated observations. A trial uses P number of particles in the particle cloud and runs for T time steps (with observations at each time step). Error statistics are computed and returned from each call to the trial function.

Prediction:

The prediction step evolves a single particle/state forward one time step according to the dynamics of the model.

Update:

The update step modifies a particle's weight according to its likelihood of yielding the current observation.

Resampling:

The resampling step chooses to keep only the top 10% of the particles with the highest weights. The weights of each particle are proportional to the likelihood of each particle yielding the current observation. The cloud is repopulated with multiples copies of these kept particles.

Program Run

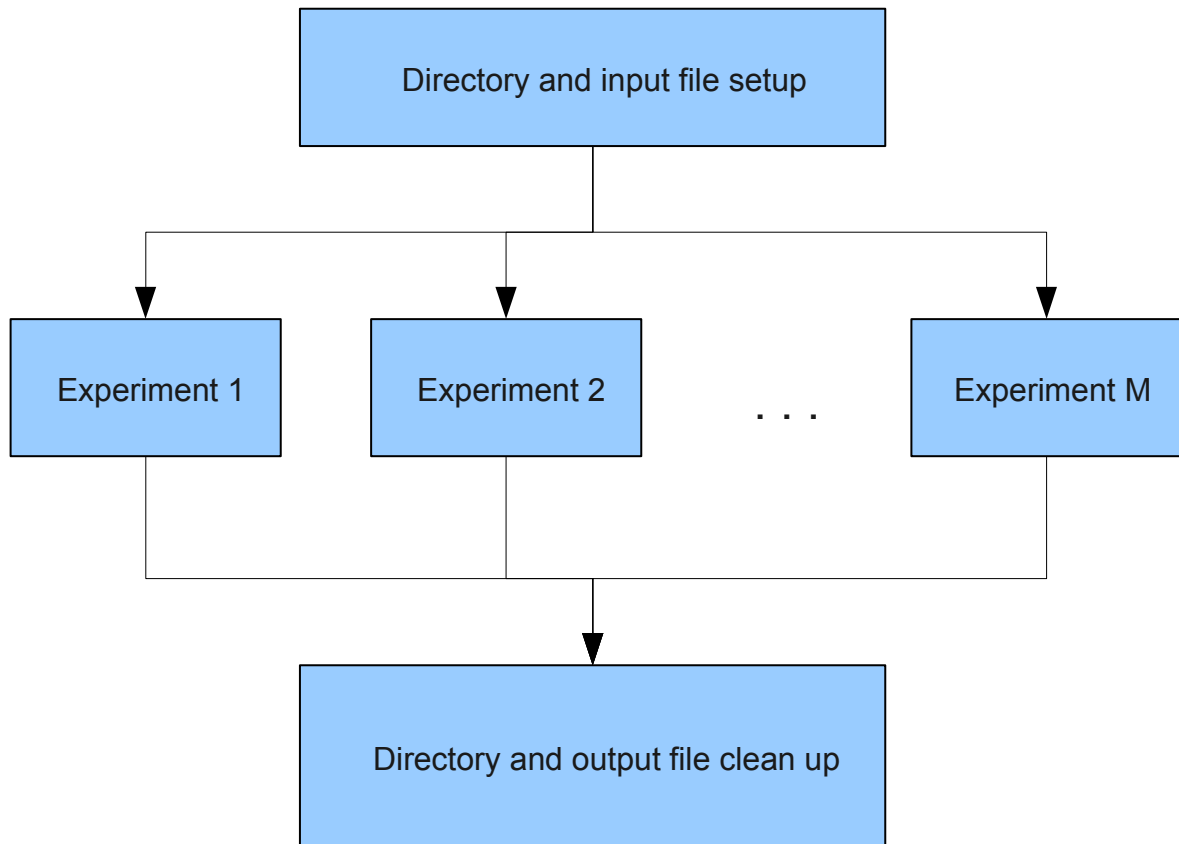


Figure 3: Program Run.

A program run is comprised of several experiments, each with its own set of parameters and initial conditions.

Experiment

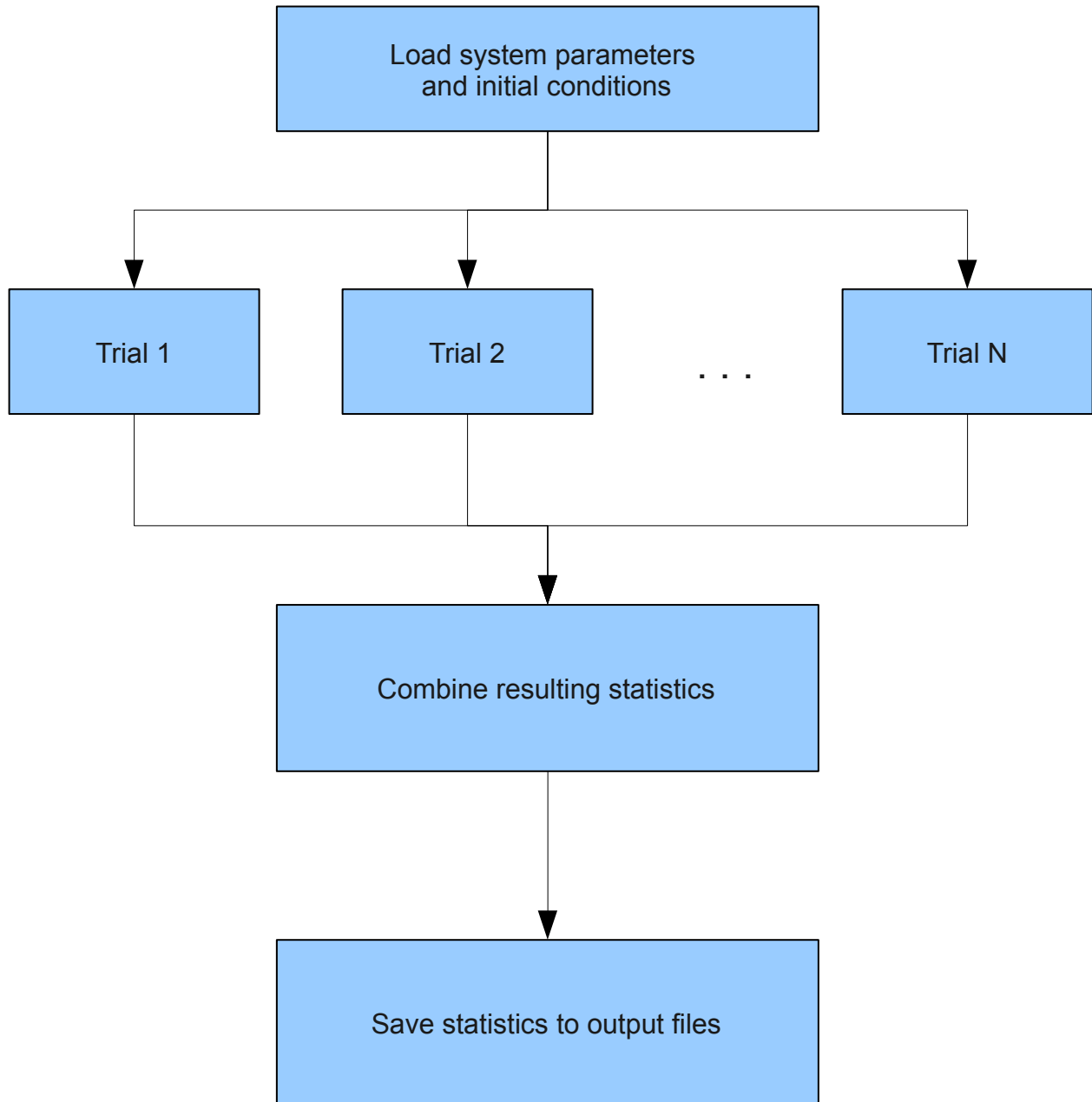


Figure 4: Experiment.

Each experiment is comprised of many trials. Each trial is independent (though they have the same input parameters and initial conditions), and can thus be run in parallel.

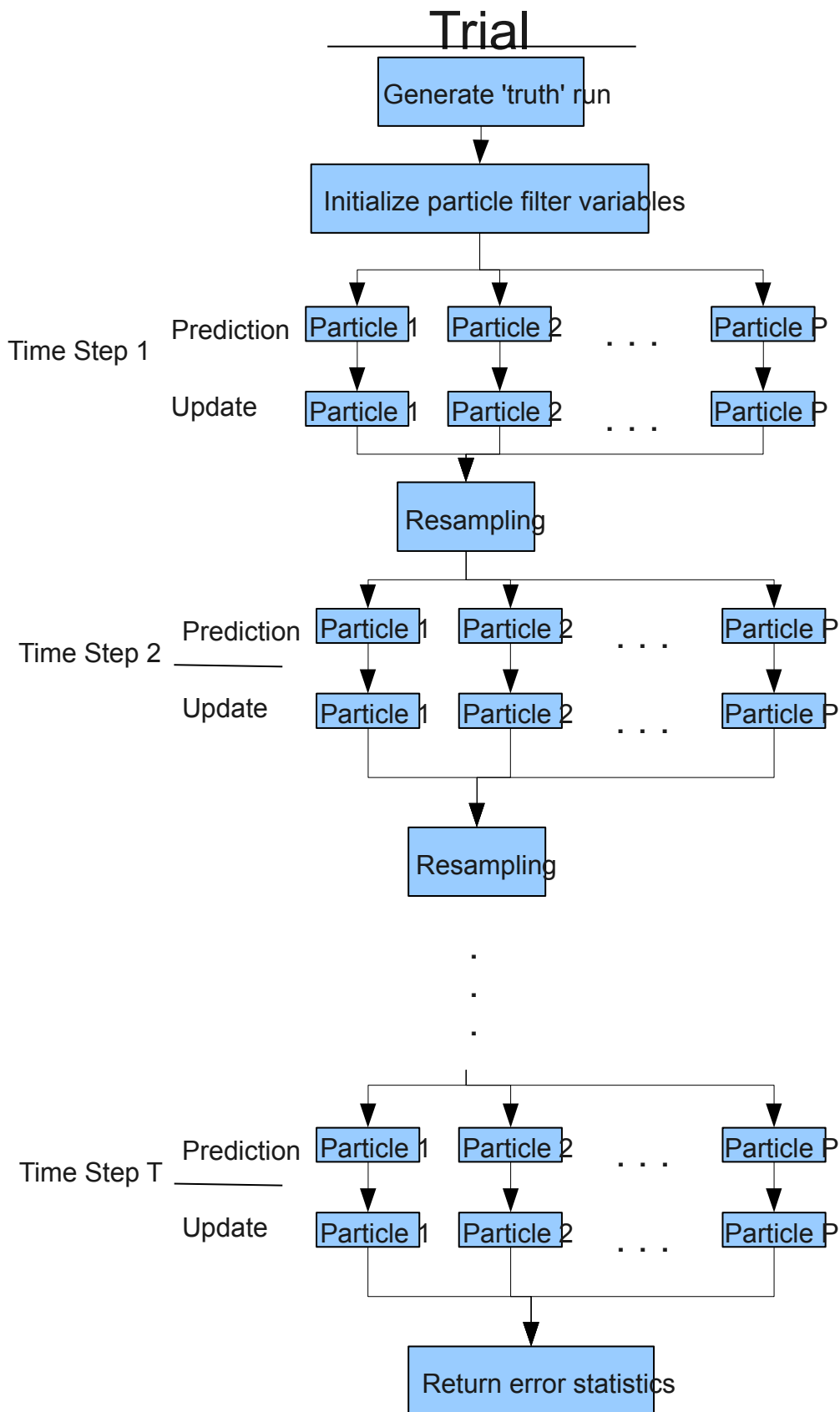


Figure 5: Trial.
 Each trial represents a single full implementation of the particle filter.

5. IMPLEMENTATION

Prior to this project we had a working implementation of the particle filter on the point-vortex model that ran sequentially in a single Matlab script. To run in parallel we first had to separate some of the components described in the previous section into independent programs. Thus, the portion of code which ran the independent trials became its own program along with a wrapper bash script which invoked this trial program. The bash script was necessary so that the trial program could be executed from the command line by the Condor jobs. So, this wrapper script is run once by each Condor job, and our Condor submit script schedules exactly one Condor job for each of the hundreds or thousands of trials that make up a single experiment.

The serial version of the program has no issues combining the results of hundreds of trial runs since all the results are already stored in convenient data structures in memory at the time of execution. However, in the parallel version, each trial script outputs files which store the error statistics and other resulting data from that specific trial run only. Thus, we had to create a post-processing Matlab script to load in all the individual trials' output and generate useful error statistics for the whole experiment. This script records the number of trials in which the particle filter failed to track the vortices' locations and computes the average error across all trials. In future work this script can be modified to compute other wanted statistics or reproduce plots and data of individual trials of particular interest.

6. RESULTS AND CONCLUSIONS

It is clear that the speedup achieved from running the trials in parallel is significant. Since most of the experiments that researchers perform contain hundreds or thousands of trial runs, the execution time being on the order of hours instead of days is important. The goal of researchers studying this data assimilation is to study at a variety of different techniques and how each one performs on different models with different parameters and initial conditions. This requires many experiment runs to capture the differences between different techniques or parameters. Thus, these researchers save a great deal of time when they can check the results of their experiments after only a couple hours instead of having to wait until the next day.

7. REFERENCES

Kuznetsov, L., Ide, K., Jones, C.K.R.T. (2003). A Method for Assimilation of Lagrangian Data. *Monthly Weather Review*, 131, 2247-2260.

Spiller, E.T., Budhiraja, A., Ide, K., Jones, C.K.R.T. (2008). Modified particle filter methods for assimilating Lagrangian data into a point-vortex model. *Physica D*, 237, 1498-1506.

APPENDIX A: Condor Pere Submission

```
Universe = vanilla
Executable = condor_job_executable.sh
Arguments = 0 $(Process)
Log = trial_condor.$(Process).log
Output = trial_condor.$(Process).out
Error = trial_condor.$(Process).err
```

```
# Queue up jobs
Queue 500
```

APPENDIX B: Condor Job Executable

```
#!/bin/bash
# Executable script used as Condor Job
# Adam Mallen

#output the date for elapsed time calculations
#start time
date

OUTPATH="output/experiment_${1}"
CURPATH="$PWD"
cd $OUTPATH
octave -q bash_trial_script.m ${1} ${2}
cd $CURPATH

#end time
date
```

APPENDIX C: Bash Setup Trial Script

```
#!/bin/bash
# Setup script to perform pre-experiment setup. Creates directories,
etc.
# Adam Mallen

# input arguments are
# (1): experiment number (parameter file number)

# enforce one argument, the number of bits
if [ $# -ne 1 ]; then
    echo "number of arguments is not exactly equal to 1"
```

```

        exit 1
    fi
# folder name is based on job number
    OUTPATH="output/experiment_${1}"
    # cleanup any prior runs with the same job number
    if [ -e $OUTPATH ]; then
        rm -r $OUTPATH
    fi

    # make fresh directories for the job output
    mkdir $OUTPATH

    # copy the script and utility files to this job's directory
    cp parameter_${1}.txt $OUTPATH #the parameter file is based on
the experiment number
    cp bash_trial_script.m $OUTPATH
    cp point_vortex_particle_filter.m $OUTPATH
    cp vortex_rotate.m $OUTPATH
    cp vortex_rotate2.m $OUTPATH
    cp SRK4.m $OUTPATH
    cp f.m $OUTPATH
    cp g.m $OUTPATH
    cp g_strengths_known.m $OUTPATH
    cp H.m $OUTPATH
    cp R.m $OUTPATH

```

APPENDIX D: Octave Trial Script

```

#! /usr/bin/octave -q

% Particle Filter driver script. Loads in parameters/initial conditions
% and runs the particle filter once with those parameters.
% Author: Adam Mallen

clear all
clc

disp('begin')

%*****
% Load parameters from file
s = argv();
if isempty(s) < 1
    u = s{1};
    in_file_num = u;
    v = s{2};
    out_file_num = v;
    parameters = load(strcat('parameter_', num2str(in_file_num), '.txt'));
    disp('Parameters used:')
    disp(parameters)
else
    file_num = 0;
end

```

```

%*****
% System & Experiment Parameters
k = 1;

num_experiments = parameters(k);
k = k+1;

num_experiments

n_t = parameters(k);
k = k+1;
n_v = parameters(k);
k = k+1;
st = 2; %dimension of a tracer's state space
sv = 3; %dimension of a vortex's state space

n = (n_t * st) + (n_v * sv); % n is the dimension of the system
m = n_t * st; %m is the dimension of the observations
d_t = parameters(k); %d_t is the time step for integration
k = k+1;
delta_t = parameters(k); %delta_t is the time step between observations
k = k+1;
total_time = parameters(k); %total simulation time
k = k+1;
n_obsrv = round(total_time/delta_t) + 1;%num is the number of observations
n_s = delta_t/d_t; %number of d_t steps per delta_t step
N = (n_obsrv - 1)*n_s + 1; %number of d_t steps in the total time
P = parameters(k); %the number of particles in the cloud
k = k+1;
keep = round(P/10); %the number of particles to keep during resampling

theta = parameters(k); %theta is the st. dev. of the observation noise
k = k+1;
theta_a_squared = parameters(k);%scaling factor to handle numerical issues
%with the fitness function R

k = k+1;
sigma = parameters(k); %sigma^2 is the variance of the stochastic
%component of the SDE

k = k+1;
alpha = parameters(k); %alpha is the st. dev. of the initial
%distribution of vortex strengths

k = k+1;

beta = parameters(k); %beta is the st. dev. of the perturbation to
% the vortex strengths upon resampling.

k = k+1;
strengths_known = parameters(k); %boolean value to determine if vortex
%strengths are known to the particles.

k = k+1;

%initial tracer locations
tracers = zeros(0,1);
for i = 1:n_t
    %Tracer starting x coordinate
    tracers = cat(1,tracers,parameters(k));
    k = k+1;
    %Tracer starting y coordinate

```

```

        tracers = cat(1,tracers,parameters(k));
        k = k+1;
end

%initial vortex locations and strengths
vortices = zeros(0,1);
for i = 1:n_v
    %Vortex starting x coordinate
    vortices = cat(1,vortices,parameters(k));
    k = k+1;
    %Vortex starting y coordinate
    vortices = cat(1,vortices,parameters(k));
    k = k+1;
    %Vortex strength
    vortices = cat(1,vortices,parameters(k));
    k = k+1;
end

%initial state
z_0 = [tracers;vortices]

%*****
% Experiment parameters
plot_flag = 0;
seed = 0;
%randn('state',1);      %set the random seed

%Number of failures per 100 experiments
N_f = 0;
%avg error in vortex positions
d = 0;

%*****
% Perform a single experiment
tt = 0:delta_t:total_time;
ttd = 0:d_t:total_time;

errors = zeros(n,N,num_experiments);
variances = zeros(n,N, num_experiments);
euclidean_errors = zeros(n_t + n_v,N,num_experiments);
euclidean_analysis_errors = zeros(n_t + n_v,n_obsrv,num_experiments);

disp('Begin simulation')

    %Set the random seed to this experiment number
    seed = sum(100*clock);

    % Run the particle filter!
    [error,euc_error,euc_analysis_err,var,truth,truth_rotated,obsrv,obsrv_rot
ated,cloud_mean,cloud_mean_rotated] = point_vortex_particle_filter(...
    seed,plot_flag,n_v,n_t,n,m,...
    total_time,d_t,delta_t,n_obsrv,n_s,N,...

```

```

        P,keep,theta,theta_a_squared,sigma,alpha,beta,strengths_known,...
        z_0);

disp('after simulation')
disp(' ')

errors(:,:,i) = error;
variances(:,:,i) = var;
euclidean_errors(:,:,i) = euc_error;
euclidean_analysis_errors(:,:,i) = euc_analysis_err;

% Check if the particle filter failed on this simulation
fail = 0;
num_fail_vortex = zeros(n_v,1);
avg_error_for_this_simulation = 0;
for j = 1:n_v
    num_fail_vortex(j) = length(find(euc_analysis_err(n_t + j,:) >= 1));
    avg_error_for_this_simulation = avg_error_for_this_simulation +
sum(euc_error(n_t + j,:),2)/(N);
end

if (sum(num_fail_vortex) > 0)
    fail = 1;
    N_f = N_f + 1;
end
if (plot_flag == 1)
    figure(i)
    hold off
    plot(truth_rotated(1,:),truth_rotated(2,:), 'k');
    hold on
    plot(truth_rotated(3,:),truth_rotated(4,:), 'r');
    plot(truth_rotated(6,:),truth_rotated(7,:), 'r');
    plot(cloud_mean_rotated(1,:),cloud_mean_rotated(2,:), 'b');
    plot(cloud_mean_rotated(3,:),cloud_mean_rotated(4,:), 'm');
    plot(cloud_mean_rotated(6,:),cloud_mean_rotated(7,:), 'm');
    plot(obsrv_rotated(1,:),obsrv_rotated(2,:), 'gs');
    hold off
end

% Compute avg vortex error for this simulation
avg_error_for_this_simulation = avg_error_for_this_simulation/(n_v);

d = avg_error_for_this_simulation;

d_avg = d;

% Save relevent data and statistics
seed_save_file_name = strcat('seed_', num2str(out_file_num), '.mat');
err_save_file_name = strcat('err_', num2str(out_file_num), '.mat');
d_save_file_name = strcat('d_', num2str(out_file_num), '.mat');
N_f_save_file_name = strcat('N_f_', num2str(out_file_num), '.mat');
truth_save_file_name = strcat('truth_', num2str(out_file_num), '.mat');
truth_rotated_save_file_name = strcat('truth_rotated_',
num2str(out_file_num), '.mat');

```

```

obsrv_save_file_name = strcat('obsrv_', num2str(out_file_num), '.mat');
obsrv_rotated_save_file_name = strcat('obsrv_rotated_',
num2str(out_file_num), '.mat');
cloud_mean_save_file_name = strcat('cloud_mean_', num2str(out_file_num),
'.mat');
cloud_mean_rotated_save_file_name = strcat('cloud_mean_rotated_',
num2str(out_file_num), '.mat');
euclidean_error_save_file_name =
strcat('euclidean_error_', num2str(out_file_num), '.mat');

save(seed_save_file_name, 'seed')
save(err_save_file_name, 'error')
save(d_save_file_name, 'd')
save(N_f_save_file_name, 'N_f')
save(truth_save_file_name, 'truth')
save(truth_rotated_save_file_name, 'truth_rotated')
save(obsrv_save_file_name, 'obsrv')
save(obsrv_rotated_save_file_name, 'obsrv_rotated')
save(cloud_mean_save_file_name, 'cloud_mean')
save(cloud_mean_rotated_save_file_name, 'cloud_mean_rotated')
save(euclidean_error_save_file_name, 'euc_error');

```

APPENDIX E: Octave Cleanup Script

```

#!/usr/bin/octave -q

% Post-processing script. Loads all results from individual trials and
% computes relevent statistics for the experiment. Then, the script
% saves the experiment statistics and other relevent data from the trials.
% Authors: Adam Mallen and Erin Soderberg

% get experiment number and number of trials as input %
s = argv();
if isempty(s) < 1
    u = s{1};
    experiment_number = u;
    v = s{2};
    num_trials = v;
else
    file_num = 0;
end

experiment_number = str2num(experiment_number)
num_trials = str2num(num_trials)

seeds = zeros(1,num_trials);
ds = zeros(1,num_trials);
N_fs = 0;

% Loop through all trials and load in their individual results %
for i = 1:num_trials

out_file_num = i - 1;

seed_save_file_name = strcat('seed_', num2str(out_file_num), '.mat');
err_save_file_name = strcat('err_', num2str(out_file_num), '.mat');

```

```

d_save_file_name = strcat('d_', num2str(out_file_num), '.mat');
N_f_save_file_name = strcat('N_f_', num2str(out_file_num), '.mat');
truth_save_file_name = strcat('truth_', num2str(out_file_num), '.mat');
truth_rotated_save_file_name = strcat('truth_rotated_',
num2str(out_file_num), '.mat');
obsrv_save_file_name = strcat('obsrv_', num2str(out_file_num), '.mat');
obsrv_rotated_save_file_name = strcat('obsrv_rotated_',
num2str(out_file_num), '.mat');
cloud_mean_save_file_name = strcat('cloud_mean_', num2str(out_file_num),
'.mat');
cloud_mean_rotated_save_file_name = strcat('cloud_mean_rotated_',
num2str(out_file_num), '.mat');
euclidean_error_save_file_name =
strcat('euclidean_error_', num2str(out_file_num), '.mat');

seed_struct = load(seed_save_file_name);
seed = seed_struct.seed;
error_struct = load(err_save_file_name);
error = error_struct.error;
d_struct = load(d_save_file_name);
d = d_struct.d;
N_f_struct = load(N_f_save_file_name);
N_f = N_f_struct.N_f;
truth_struct = load(truth_save_file_name);
truth = truth_struct.truth;
truth_rotated_struct = load(truth_rotated_save_file_name);
truth_rotated = truth_rotated_struct.truth_rotated;
obsrv_struct = load(obsrv_save_file_name);
obsrv = obsrv_struct.obsrv;
obsrv_rotated_struct = load(obsrv_rotated_save_file_name);
obsrv_rotated = obsrv_rotated_struct.obsrv_rotated;
cloud_mean_struct = load(cloud_mean_save_file_name);
cloud_mean = cloud_mean_struct.cloud_mean;
cloud_mean_rotated_struct = load(cloud_mean_rotated_save_file_name);
cloud_mean_rotated = cloud_mean_rotated_struct.cloud_mean_rotated;
euc_error_struct = load(euclidean_error_save_file_name);
euc_error = euc_error_struct.euc_error;

seeds(i) = seed;
ds(i) = d;
N_fs(i) = N_f;
total_N_fs = N_fs + N_f;

end

% save Experiment statistics and relevent data %
d_avg = mean(ds)
percent_failure = total_N_fs/num_trials

ds_save_file_name = strcat('ds_', num2str(experiment_number), '.mat');
N_fs_save_file_name = strcat('N_fs_', num2str(experiment_number), '.mat');
d_avg_save_file_name = strcat('d_avg_', num2str(experiment_number), '.mat');
seeds_save_file_name = strcat('seeds_', num2str(experiment_number), '.mat');
fail_rate_save_file_name =
strcat('fail_rate_', num2str(experiment_number), '.mat');

```

```
save(ds_save_file_name, 'ds');
save(N_fs_save_file_name, 'N_fs');
save(d_avg_save_file_name, 'd_avg');
save(seeds_save_file_name, 'seeds');
save(fail_rate_save_file_name, 'percent_failure');
```

APPENDIX F: Bash Cleanup Trial Script

```
#!/bin/bash
# Clean up script to perform post experiment processing.
# Adam Mallen and Erin Soderberg

# input arguments are
# (1): experiment number (parameter file number)
# (2): number of trials in this experiment

# enforce one argument, the number of bits
if [ $# -ne 2 ]; then
    echo "number of arguments is not exactly equal to 2"
    exit 1
fi
# folder name is based on job number
CURPATH="$PWD"
OUTPATH="output/experiment_${1}"
NUMTRIALS=${2}

rm trial_condor.*

cp octave_clean_up.m $OUTPATH/
cd $OUTPATH
octave -q octave_clean_up.m ${1} $NUMTRIALS

cd $CURPATH

#remove copied utility files
rm $OUTPATH/bash_trial_script.m
rm $OUTPATH/point_vortex_particle_filter.m
rm $OUTPATH/vortex_rotate.m
rm $OUTPATH/vortex_rotate2.m
rm $OUTPATH/SRK4.m
rm $OUTPATH/f.m
rm $OUTPATH/g.m
rm $OUTPATH/g_strengths_known.m
rm $OUTPATH/H.m
rm $OUTPATH/R.m
```