NLNG: A R PACKAGE FOR STATE SPACE MODELS

by

Joey Hartling

Submitted in partial fulfillment of the requirements
for the degree of Master of Statistics

at

Dalhousie University
Halifax, Nova Scotia
August 2019

© Copyright by Joey Hartling, 2019

Table of Contents

List of Tables . . . **iv**

List of Figures . . . **v**

Abstract . . . **vii**

List of Abbreviations and Symbols . . . **viii**

Chapter 1 Introduction . . . **1**
1.1 Motivating Example . . . 3
1.1.1 2D Random Walk Model . . . 5
1.2 Rise of Programming Tools for State Space Models . . . 6
1.2.1 State Space Models Tools in R . . . 6
1.2.2 Need For Nonlinear and Non-Gaussian Tools . . . 10
1.3 Thesis Overview . . . 11

Chapter 2 Background: State Space Theory . . . **13**
2.1 The General State Space Model . . . 13
2.2 Filtering Techniques . . . 14
2.2.1 The Kalman Filter . . . 16
2.2.2 The Particle Filter . . . 18
2.2.3 Sequential Importance Resampling (SIR) . . . 19
2.3 Smoothing Techniques . . . 21
2.3.1 Kalman Smoother . . . 22
2.3.2 Particle Smoother . . . 23
2.4 Parameter Estimation . . . 25
2.4.1 Maximum Likelihood Estimation (MLE) . . . 26
2.4.2 State Augmentation . . . 28
2.4.3 Multiple Iterative Filtering (MIF) . . . 30

Chapter 3 An R Package: nLnG . . . **33**
3.1 Package Creation . . . 35
3.1.1 Introduction . . . 35
3.1.2 Linking Complied Code . . . 36
3.1.3 Help Pages . . . 37
3.1.4 Package Management . . . 37
3.1.5 R-Studio . . . 38
3.1.6 Error Checking . . . 39
3.1.7 Installation . . . 40
3.1.8 Programming for Package Creation . . . 41
3.1.9 nLnG Package Creation Details . . . 44

3.2 Defining an nLnG State Space model object and Functions 47
3.2.1 Example Problem: State Model Definition 50
3.3 Creating an nLnG Object and Simulating Data (simulate) 51
3.4 The Kalman Filter (kFilter) 56
3.5 The Particle Filter (pFilter) 62
3.6 The Kalman Smoother (kSmoother) 70
3.7 The Particle Smoother (pSmoother) 74
3.8 MLE Methods 80
3.9 State Augmentation (sAugmentation) 83
3.10 Multiple Iterative Filtering (mif) 88

Chapter 4 Observed Data Analysis with nLnG 101
4.1 Data Introduction 101
4.2 Model Selection and Parametrization 104
4.2.1 Model Selection 104
4.2.2 Parametrization 107
4.3 Parameter Estimation 109
4.3.1 Preparing the Observed Data nLnG Object 109
4.3.2 Simulation Testing 111
4.4 Filtering and Smoothing Results 118
4.4.1 Filtering 119
4.4.2 Smoothing 122

Chapter 5 Conclusion 126
5.1 Package Creation 126
5.2 Observed Data Analysis 127
5.3 Future Work 129

Appendix A Code Appendix 130

Appendix B Plots and Figures 150

Appendix C Comparisons Appendix 175
C.1 Filter Comparison 176
C.2 Smoother Comparison 178

Appendix D List of Algorithms 181

Bibliography 182

List of Tables

1.1 R Packages Not Needing Object Summary 8

1.2 R Packages Function Object Needed Summary 9

1.3 Library Support List 11

3.1 State and Parameter linking to `nLnG` variables 34

3.2 S3 Class and Methods 42

3.3 S4 Class and Methods 43

3.4 nLnG and pomp Comparison 45

3.5 MLE Parameter Estimate Results 82

3.6 MIF Estimated Parameter Values 100

4.1 Real Problem: State and Parameter Initial Values 112

4.2 MIF:Estimate Values 118

List of Figures

1.1 Observed Animal track 5

3.1 Plotted Simulated data 55
3.2 Plotted Kalman filter Results 61
3.3 Plotted Effective Sample Size 68
3.4 Filtered Results for `pFilter` 69
3.5 Plotted Kalman Smoother Mean 73
3.6 Plotted Particle Smoother Mean 79
3.7 Plotted State Augmentation Parameter Estimation 88
3.8 MIF Filter Diagnostics For States and Parameter 97
3.9 MIF Convergence Diagnostics For States and Parameter . . . 98

4.1 Plot of the Real Animal Data Set 104
4.2 Plot of the Real Animal Data MIF Results 1 115
4.3 Plot of the Real Animal Data MIF Results 2 117
4.4 Real Data: Effective Sample Size Plot 120
4.5 Real Data: Particle Filter Results 121
4.6 Real Data: Particle Smoother Results 123
4.7 Real Data: Particle Smoother vs Particle Filter Results 125

B.1 Plotted Simulated data using `plot` 150
B.2 2D Kalman Filter Mean w/ 95%CI Plot 151
B.3 Longitude Kalman Filter Mean w/ 95%CI Plot 151
B.4 Latitude Kalman Filter Mean w/ 95%CI Plot 152
B.5 Kalman Filter: Mean and Variance Plots 153
B.6 2D Particle Filter Quantile Plot 154

B.7	Longitude Particle Filter Shaded Quantile Plot	155
B.8	Latitude Particle Filter Shaded Quantile Plot	156
B.9	Particle Filter: Means and Variance Plots	157
B.10	2D Particle Filter Mean Plot	158
B.11	2D Kalman Smoother Mean w/ 95%CI Plot	159
B.12	Longitude Kalman Smoother Mean w/ 95%CI Plot	160
B.13	Latitude Kalman Smoother Mean w/ 95%CI Plot	161
B.14	Kalman Smoother: Mean and Variance Plots	162
B.15	2D Particle smoother 95% Quantile plot	163
B.16	Longitude Particle smoother 95% Quantile plot	164
B.17	Latitude Particle smoother 95% Quantile plot	165
B.18	Particle Smoother: Mean and Variance Plots	166
B.19	2D Particle Smoother Mean Plot	167
B.20	Longitude PF Median vs. KF Mean Plot	168
B.21	Longitude PF Median vs. KF Mean Plot	168
B.22	Longitude PF Median vs. PS Median Plot	169
B.23	Latitude PF Median vs. PS Median Plot	170
B.24	PF vs. PS: Mean and Variance Plots	171
B.25	Longitude KF Mean vs. KS Mean Plot	172
B.26	Latitude KF Mean vs. KS Mean Plot	173
B.27	KF vs. KS: Mean and Variance Plots	174
C.1	2D PF vs. KF Comparison Plot	177
C.2	PF vs. KF: Mean and Variance Plots	178
C.3	2D PS vs. KS Comparison Plot	179
C.4	PS vs. KS: Mean and Variance Plots	180

Abstract

State space models require the ability to perform filtering, smoothing and prediction during analysis. To perform these procedures fairly complex computational algorithms are required. There is a consequent need for software tools to facilitate the implementation of state space models. One of leading choices for computation data analysis is the statistical programming language, R. And a key area where analysis tools are lacking is for advanced state space models. This thesis outlines the development of a comprehensive toolkit for nonlinear and non-Gaussian state space models: the nLnG package.

List of Abbreviations and Symbols

`nLnG`	systex for R package names
`pFilter`	systex for R code segments
NLNG	nonlinear non Gaussian
AR	autoregressive
ARMA	autoregressive moving average
SSMs	state space models
2D	two dimensional
RWM	random walk model
CRW	correlated random walk
BRW	biased random walk
EKF	extended Kalman filter
MIF	multiple iterative filtering
MCMC	Markov chain Monte Carlo
SMC	sequential Monte Carlo
SIS	sequential importance sampling
SIR	sequential importance resampling
PDF	probability density function
MLE	maximum likelihood estimation
IDE	integrated development environment
CRAN	Comprehensive R Achieve Network
OOP	object-oriented programming
RNG	random number generator
OTN	ocean tracking network
$\mathbf{A}'$	transpose of matrix $\mathbf{A}$
$\mathbf{A}^{-1}$	inverse of matrix $\mathbf{A}$
$L_t(\theta)$	likelihood function at time t
$\mathbf{l_t}(\theta)$	log likelihood at time t for theta
$N(\mu, \Sigma)$	Gaussian distribution with mean μ and covariance Σ

$U(a,b)$	Uniform distribution between a and b
$wC(0,c_t)$	wrapped Cauchy distribution at concentration c_t
t	time index
$\propto$	proportional according to
$\sim$	distributed according to
$\approx$	approximately equal to
Δ	change in time
$\mathbf{x}_t$	state vector at time t
$\mathbf{v}_t$	state error vector at time t
$d(\cdot)$	state dynamics function
θ_t	dynamical parameters at time t
$\mathbf{y}_t$	observation vector at time t
$\mathbf{e}_t$	observation error vector at time t
$h(\cdot)$	observation operation function
ϕ_t	observation operator parameter at time t
$p(\mathbf{x}_t\|\mathbf{y}_t)$	filter probability density at time t
$p(\mathbf{x}_t\|\mathbf{y}_{t-1})$	predictive probability density at time t
$p(\mathbf{y}_t\|\mathbf{x}_t)$	likelihood of observations at time t
$p(\mathbf{x}_t\|\mathbf{x}_{t-1})$	model "transition" density at time t
$\mathbf{Q}_t$	system noise error covariance matrix at time t
$\mathbf{R}_t$	observation noise error covariance matrix at time t
$\{\cdot\}$	ensemble
Np	number of particles
$\hat{\mathbf{x}}_{t\|t-1}$	prediction mean at time t
$\mathbf{M}_t$	prediction error covariance matrix at time t
$\hat{\mathbf{x}}_{t\|t}$	filter mean at time t
$\mathbf{P}_t$	filter error covariance matrix at time t
$\mathbf{K}_t$	Kalman gain function
$\mathbf{K}_t^*$	Kalman smoother gain function
$\mathbf{P}_{N\|N}$	smoother variance at time N given N observations
$\mathbf{F}_t$	innovation at time t

$\mathbf{e}_t^{\theta}$	parameter error vector at time t
σ	mif starting parameter values
$w_t^{(i)}$	i^{th} member of the weighted sample
$\tilde{\mathbf{x}}_t$	smoother posterior density sample at time t
γ_t	correlation magnitude
$\dot{\mathbf{x}}_t$	augmented state vector at time t
$\dot{\mathbf{e}}_t$	augmented parameter vector at time t

Chapter 1

Introduction

Throughout the field of statistics it is not uncommon for the problem being studied to become larger or more complex than one may readily handle without computational support such as analysis packages. There are many different programs available to aid with this computational support and it has become important for statisticians to have and make such resources available in the community. In computer science, libraries and packages act as tools for programs and these tools work as collections of resources and can consist of pre-written code, subroutines, classes, types, etc; each part adds functionality to a program's base structure. The technique of using libraries and packages has also become very popular in the field of computational statistics when looking to add the needed statistical functionality.

There are many areas of interest in the field of statistics, so it only makes sense that the collections are usually grouped together by these areas of interest into libraries and packages. One such area where this problem occurs is that of analysing dynamical systems, more commonly known in the statistical field as being a part of time series analysis called; State Space Models (SSMs). SSMs solutions tend to be computationally heavy and rely greatly on programs to produce results for problems of interest. While there are available libraries and packages for SSMs currently available, these libraries and packages do not cover a full set of procedures. One particular procedure worth making available for SSMs is that of smoothing for Nonlinear Non-Gaussian (NLNG) systems. These types of SSMs can be used in modelling weather systems, stock markets and tracking animal behaviour (as we will show), and a smoothing procedure is a commonly needed procedure when working with SSMs.

A SSM is made up of a two part system of equations: the state (or process) equation; and the observation (or measurement) equation. The state equation advances

the system state forward through time, as the observation equation makes measurements of the state available at given times. The state equation is a dynamical system model that usually describes some real world phenomenon and the observation equation describes the statistical properties of equipment/tools used to measure such phenomena. An example that a SSM system could be used to describe is weather patterns, where the location and how that weather system acts is described by the state equation and the equipment used to locate that weather system is described by the observation equation. Both the state and observations equations/models involve random error (noise). In the weather system example the error/noise in the state equation would be equivalent to the difference in expected location of the weather system from a movement the model may not account for. The error/noise in the observation equation would be equivalent to the measurement error used by the satellite to track the weather system. The goal in using SSMs is to make optimal estimates of the full system state (or true system state). Optimal estimates of the true state are produced by using the models, and the noisy observations; each contains information about the true state of the system.

These developed libraries for analysing SSMs range across many of the available language platforms for programming. One of these platforms is the statistical programming language R available from the Comprehensive R Archive Network (CRAN) (R Core Team, 2014). R is a fully available open source independent program that encourages users to create their own libraries/tools. These tools then can be submitted and shared with the rest of the user community through CRAN or other contribution sites, like R Forge (https://r-forge.r-project.org/). With many tool packages already having been created for R (a number of which offer SSMs analysis methods), its importance within the statistical community has grown quickly and it has become the program of choice for many statisticians. This is because R is free to obtain, has a large user base and it is verified/maintained by well known statisticians in the community. It makes sense to use R for any new development of tools for statistical programming, since any developments are shared through one of the mentioned methods above. Contributions to the larger community as a whole arises, as the software becomes freely available for all to use.

R has several packages for analysing SSMs. Most of these packages concentrate on working with just SSMs that are of the linear Gaussian type. An area of SSMs that still needs development for R is working with NLNG types of SSMs. The general types of SSM problems that need to be solved are still the same in both the linear Gaussian and the NLNG classes of SSMs; they consist of filtering, smoothing, prediction and parameter estimation. Filtering requires that we can predict the state at the end of the observation time interval, smoothing asks that we can predict the state in the middle of the observation time interval and prediction wants us to predict the state beyond the end of the observation time interval. Parameter estimation requires that we can predict values for the model parameters and/or error structures that describe the error/noise. In the R package developed in this thesis we will look to enhance the ability to solve these problems for NLNG type of SSMs. The new package will contain methods for solving the filtering and parameter estimation problems, and is the first to include a method for solving the smoothing problem.

1.1 Motivating Example

One interesting example that occurs in marine ecology is that of tracking the location of animal movements as they occur in their ocean habitat. The problem is that we are unable to directly monitor animals for long periods of time without influencing their behaviour in some manner. The ability to analyse movement tracks has been aided greatly by technologies like satellite, archival and harmonic radar tags (electronic chips usually attached to the animal) which are able to produce measurements of location. These methods of measurements are susceptible to errors or deviations from the true location of the animal, this is known as observation error. The use of these technologies has allowed the determination of animal movements over great distances (Roland et al., 1996; Bergman et al., 2000; Block et al., 2001). Accounting for errors, such as deviations in recording measurements from equipment can now be incorporated into the observation equation. We want to determine the true state

location from the measurement observations and/or we also wish to infer the animal's behaviour through parameter estimation. This animal tracking problem is now exactly the type of situation we wish to solve using SSMs tools.

Like all SSMs the animal tracking process is represented by a 2 part equation system, first the animal movements are represented through a state equation and second, a recordable measurement of the location for the animal is then produced by the observation equation relating the system state. The observation equation is generally just relative to the measurement to the true state by a relation function and error term accounting for the shortcomings in the measurement equipment. It is the state equation that controls the animals movement and behaviours. The state equation must be defined in a way that mimics true animal movement. One way to think about this is graphically. The movement locations are much like moving around a xy-plane space (sometimes a xyz-space is used to add depth for an animal). Instead of xy coordinates, we record locations in terms of latitude and longitude (see Figure 1.1 for an example animal movement track plot). We therefore need a 2-dimensional (2D) system state to represent the animal's position. It has been shown that animal movement paths are statistically consistent with Random Walk Models (RWM), in particular the correlated random walk model (CRW) or biased random walk model (BRW) (Kareiva and Shigesada, 1983; Turchin, 1998). This leads us to use the 2D RWM for representing animal movements as our state model.

Animal tracking problems require that one be able to reproduce the tracks by simulation, perform filtering and smoothing on the noisy observations and get estimates of parameters. It has been shown that filtering and smoothing methods are able to estimate true states and parameters for linear Gaussian SSMs (Nielsen et al., 2006; Johnson et al., 2008). Since animal tracking data and models are typically not linear and Gaussian in nature, methods have been developed for NLNG SSMs to account for this problem. The first solution is using Markov Chain Monte Carlo (MCMC) methods to estimate true states and parameters (Jonsen et al., 2005; Patterson et al., 2008). The second approach, less popular, but quickly advancing is to use Sequential Monte Carlo (SMC) methods (Ionides et al., 2006, 2011; Dowd and Joy, 2011) to get estimates of true states and parameters. This approach presents advantages for both

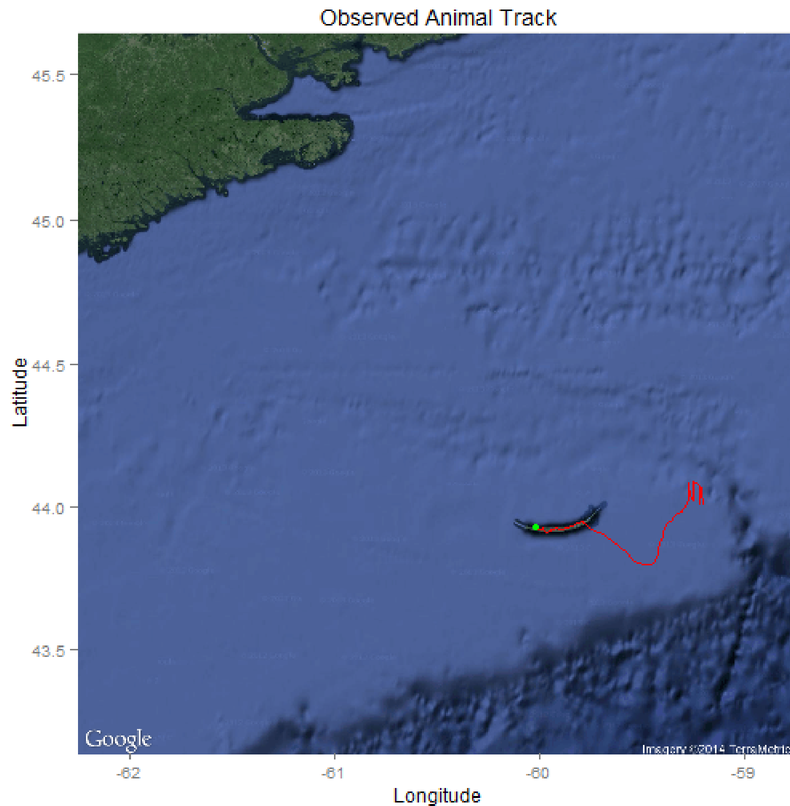


Figure 1.1: Recorded observations from a Atlantic grey seal on the Scotian shelf, supplied by the Ocean Tracking Network at Dalhousie University, Halifax, NS. The initial animal position is marked by the green point, while the movement path is the highlighted red line.

the linear Gaussian, as well as the NLNG type SSMs.

1.1.1 2D Random Walk Model

The RWM is a special case of time series models called autoregressive models, in the animal tracking case we will be using a 2D, or bivariate version. The state $\mathbf{x}_t$ is represented by a 2×1 random vector with elements corresponding to the longitude and latitude positions at the current time, t. The RWM here is comprised of an initial starting point plus white noise and is of the following form:

$$\mathbf{x}_t = \delta + \mathbf{x}_{t-1} + \mathbf{v}_t \tag{1.1}$$

where the $\mathbf{x}_t$ is the state vector at time t, δ represents the drift term and $\mathbf{v}_t$ is the state error at time t. The model contains a dynamic function that relates the animal's behaviour based on a set of parameters from the last time position to the new time position and may have a much more complex error structures than that of just white noise. For the rest of this thesis we will first produce synthetic animal tracking data from a 2D RWM with a drift term and secondly use a more complex 2D correlated RWM to model movement of an Atlantic Grey Seal from observed data. These will be used to test the SSM algorithms.

1.2 Rise of Programming Tools for State Space Models

The creation of tools for SSMs was slow to develop and this was expressed best by Durbin and Koopman (2001) when they wrote "In our opinion, the only disadvantages are the relative lack in the statistical and econometric communities of information, knowledge, and software regarding these models.". As of 2001, the lack of software available for analysing SSMs was still one of the limiting factors in working with these types of models. Since then though, there has been an explosion of software tools developed for analysing SSMs. These tools were developed using a wide variety of programs, which includes R, MATLAB, SAS, C, S-Plus, **STAMP**, **REGCMPNT**, EViews, GAUSS, Stata, **RATS**, gretl, Python, etc. Now there is obviously no longer a shortage of available tools for analysing SSMs. The problem now is what algorithms have or should be implemented in SSM tools? We shall now do a small review of the functionalities of some of the currently developed tools, focusing mainly on the tools available within R for SSMs.

1.2.1 State Space Models Tools in R

A general requirements list for packages for SSMs in R is given by Petris and Petrone (2011). They suggested that packages must offer functions for performing filtering, smoothing and forecasting (prediction) for both univariate and multivariate linear

Gaussian SSM and pared the packages down to 3 that satisfied the requirement at that time. The packages are **Dynamic System Estimation** (dse) (Gilbert, 2009), **Dynamic Linear Modelling** (dlm) (Petris, 2010) and **Kalman filters and smoothers for Exponential family state space models** (KFAS) (Helske, 2012). These are the packages that offered the most complete set of tools for working with SSMs. Petris and Petrone (2011) also presented a number of other SSM packages for R that we will cover in this review. Lastly the package **Statistical inference for partially observed Markov processes** (pomp) (King et al., 2010) is the first library or set of tools really aimed at working with SSMs of the NLNG framework and will be where we finish up the review of current R packages available.

Let us start this review with packages that do not require the user to encode the data in an object. The packages available that meet this requirement are the base stats package and the package KFAS. In Table 1.1 below, the functions for these two packages are summarized for analysing SSMs. Looking at the table; some common trends can be seen in the packages. Both packages contain a Kalman filter and Kalman smoother function and each require only that the data be of `ts` type (the base time series data type in R). When R is downloaded the base stats package includes functions for modelling, using the Kalman filter, smoothing data and making forecasts. One disadvantage of the stats package is that it does not include many functions for working with the SSM object within the package. This means that besides the included applications, additional packages will be needed to add the missing SSM applications. The package KFAS (Helske, 2012) includes a variation of Kalman filters and smoothers, functions for calculating the log-likelihood of the model and a simulation smoother.

The packages for R summarized in Table 1.2 are made in the object oriented framework and require that the data be made into an object that is usable by the functions. Table 1.2shows the functionality offered across the three packages dlm, dse and **State Space Models in R** (sspir) (Dethlefsen et al., 2012). All three packages offer at least one method for filtering and smoothing, packages dlm and dse each offer methods for forecasting, Maximum Likelihood Estimation (MLE) and modelling autoregressive moving average (ARMA) Models. All three packages have their own

Processes	base	KFAS
kalman filter	KalmanLike	kf,OOIndex
forecasting	perdict.arima	NA
smoothing	tsSmooth	ks,efsmooth
MLE/LogLik	NA	flik,eflik0
simulating	NA	simsmoother
arima	ARIMA	NA
object	ts	ts

Table 1.1: Functions avaliable for the listed state-space methods for the ts data type packages in R. Across the top the packages are listed, with the method listed down the left side. If the package doesn't contain a function NA is listed, else the function name is given for the method(s)

method of creating an object for the SSM. Looking closer at each packages' finer points; package sspir was created to give a formula language for specifying dynamic generalized models, and is an extension of the glm formulation with the coefficient allowed to be time-varying. The package requires that functions be expressed in one of two ways, the first being for representing Gaussian SSMs and second for defining SSMs into a glm-style call. Package dlm was designed for doing Bayesian and likelihood analysis of dynamic linear models (SSM). They have included some of the basic techniques such as the MCMC output analysis for performing adaptive rejection Metropolis sampling, drawing from the posterior distribution of state vectors and Gibbs sampling for the d-inverse-gamma model. Other functions included in the package are mostly for working with the data object, though they do have a function that will create a random dlm object (or more general a random SSM). The largest of the packages that has been produced for R is dse. The packages main functions are for creating objects for time input and output data, for a dse structure and for model, data and estimation information. The general methods of the package are for constructing ARMA and state space models, while the main analysis tools included are for evaluating a model, simulating from a model and for smoothing a model. It then offers methods for producing and evaluating forecasts and also methods for performing model diagnostics.

The newest package released for working with SSMs in R that supports the NLNG case models is pomp (King et al., 2010). It was created to provide inference on time

Processes	sspir	dlm	dse
kalman filter	kfilter,extended	dlmfilter	NA
forecasting	NA	dlmForecast	forecasts
smoothing	kfs,smoother	dlmsmooth	smoother
MLE/LogLik	NA	dlmLL,MLE	estMaxLik
simulating	recursion	simulate	NA
arima	NA	dlmModARMA	ARMA,toARMA
object	SS,ssm	dlm	TSdata,TSmodel

Table 1.2: Functions avaliable for the listed state-space methods for the object using packages in R. Across the top the packages are listed, with the method listed down the left side. If the package doesn't contain a function NA is listed, else the function name is given for the method(s)

series data using partially-observed Markov processes or SSMs, with the most important feature being that it was designed for nonlinear stochastic dynamical systems. pomp is the first of the tools we have looked at that openly tackled the problem of NLNG models, a key new area of development for SSMs. Notice now the difference in functionality that pomp offers compared to packages for R as summarized in Tables 1.1 and 1.2. pomp requires that the user be able to setup the object with the system models and variables. The first major challenge is that the Kalman filter is not applicable for NLNG models, so it is replaced by the particle filter, multiple iterative filtering (MIF) and a Bayesian particle filter by Liu and West (2001) in this package. They also include a function for fitting models to data using nonlinear forecasting for model prediction, as well as one to simulate from the SSM. Although not directly related to SSMs, the next functions included are trajectory matching for fitting a deterministic dynamical trajectory to a set of data and probe matching which fits a model using summary statistics to data until the agreement between model and data is maximized by some criterion. pomp has been the only SSM package for R that has continued to build upon the framework, by later adding a second MIF application (Ionides et al., 2011) and continuing modifications over time. With the development of pomp, the ability to work with SSMs in the NLNG framework has taken a step forward.

This ends the review of tools that are currently available for working with SSMs. The goal here was to build up knowledge of what tools and functionality was included

in existing software packages for SSMs. Any new creation of software should at a minimum add to the functionality offered by existing software, relevant to the analysis of SSMs. From the review, it can be seen the linear and Gaussian model case has been covered for both univariate and multivariate SSM frameworks quite thoroughly. Table 1.3 shows the existing packages in R all support working with both the univariate and multivariate models (except for the base package). Therefore any new packages for working with SSM should include this functionality to support the linear Gaussian case. The tools that should be offered in the package for the linear Gaussian model consist of filtering, smoothing, forecasting, estimating, calculating likelihoods, simulating and diagnostic tests checking. This would represent a complete set of analysis tools comparable to what current software offers and this set of tools will be outlined in the next chapters.

1.2.2 Need For Nonlinear and Non-Gaussian Tools

The tools for linear Gaussian SSMs are well defined; while for the NLNG SSMs case the opposite is true. Table 1.3 shows that support for NLNG models within R packages is not as common, as only few packages offer options. Both stochastic volatility models and dynamic generalized linear models are covered to a certain extent, as they are probably the most popular NLNG models. Package sspir offers the ability for univariate extended Kalman filtering and smoothing for the dynamic generalized linear models with Poisson or Binomial distributions, while package KFAS offers approximate Kalman filtering and smoothing for these model types. The release of pomp has done much to provide a set of tools for working with NLNG SSMs that was not available previously. The ability to work with NLNG models has seen some improvement with the release of pomp. The package pomp offers the user filtering, estimation and forecasting methods to work with NLNG SSMs, however the overall framework of working with NLNG SSMs still has room for improvement. The new package nLnG created for this thesis will look to develop one of these methods in particular, smoothing for NLNG SSMs. The smoothing method that will be used in

Library	LG	MLG	NLNG
R-base v2.15.0	Y	N	N
R-sspir v0.2.8	Y	Y	Y
R-dse v2012.4-1	Y	Y	N
R-KFAS v0.6.1	Y	Y	Y
R-dlm v1.1-2	Y	Y	N
R-pomp v0.42-4	Y	Y	Y

Table 1.3: Libraries that support the model type are marked with a ”Y” and model type not supported are marked with an ”N”. Where LG - Linear Gaussian Models, MLG - Multivariate Linear Gaussian Models and NLNG - NonLinear and Non-Gaussian Models

the packages and in this thesis will be the particle smoother of Godsill et al. (2004). This method has not been tackled for the NLNG case to date.

1.3 Thesis Overview

This thesis is structured in the following manner. Chapter 2 will act as a background chapter for SSMs, in which all the theory for the general SSM filtering, smoothing and parameter estimation problems will be introduced for the algorithms used for the new package. The following topics will be covered:

- General SSM
- Kalman Filter
- Particler Filter
- Kalman Smoother
- Particler Smoother
- Maximun Likelihood Estimation
- State Augmentation
- Multipler Iterative Filtering

In Chapter 3, we start by illustrating to the reader the details of creating new packages for R and explain the methods used to create the new package nLnG. We then move into a demonstration of all the functionality of the new software using synthetic RWM data that represents an animal pathway. We then show the user how to encode SSMs with nLnG, as well as the methods for extracting and changing parts of the encoded model. Next the filters and smoothers are both demonstrated for nLnG and the reader is shown how to set up the functions and process the object for the results from the procedures. An appendix showing how the results for the synthetic RWM data are equivalent under both the linear Gaussian and NLNG filtering and smoothing methods of state estimation can be found in Appendix C. Finally the methods for parameter estimation will be illustrated using one of the parameters from the synthetic RWM used throughout the chapter. Chapter 4 will use the package nLnG to analyse a real set of observed animal tracking data, showing the more general approach to working with these types of problems with the new software. This thesis will conclude with a summary of final thoughts in Chapter 5, as well with suggestions for future work for the software.

Chapter 2

Background: State Space Theory

This chapter will present the general background information for the SSM concepts being used in the new `nLnG` package. It can be broken down into three main parts. The first part is filtering techniques, the second part is smoothing techniques and the third part is parameter estimation methods. In each of these sections we will first introduce the general algorithm for the technique, then detail each method used for the procedure in `nLnG` (plus any methods needed for supplementary support). Below, we will start by introducing the theory for SSMs in term of the general model form.

2.1 The General State Space Model

The general SSM is made up of a two part equation system: the state (or process) equation, and the observation equation that relates the observations to the true state. They are defined as follows:

$$\mathbf{x}_t = d(\mathbf{x}_{t-1}, \theta) + \mathbf{v}_t, \tag{2.1}$$

$$\mathbf{y}_t = h(\mathbf{x}_t, \phi) + \mathbf{e}_t, \tag{2.2}$$

where

$\mathbf{x}_t$: state vector at time t

$\mathbf{v}_t$: state error vector at time t

$d(\cdot)$: state function describing the dynamics

θ: the dynamical parameters

$\mathbf{y}_t$: observation vector at time t

$\mathbf{e}_t$: observation error vector at time t

$h(\cdot)$: observation operator function that relates $\mathbf{x}_t$ to $\mathbf{y}_t$

ϕ: observation operator parameters

The state (process) equation, $\mathbf{x}_t$, is a difference equation, where one or more of the variables are stochastic making it an stochastic difference equation. The process $\mathbf{x}_t$ is also a Markov process meaning that $\mathbf{x}_t$ only depends on the state at the previous time, $\mathbf{x}_{t-1}$ using the dynamical state function, $d(\cdot)$, and dynamical parameter, θ, that controls how the system evolves through time. The state, $\mathbf{x}_t$, then is related to the observation equation, $\mathbf{y}_t$, by the observation operator function, $h(\cdot)$, and the operator parameter, ϕ. The observation equation, $\mathbf{y}_t$, tends to resemble regression slightly, though regression approaches would only treat the observation equation alone and not account for the influence from the state equation. The goal with SSM solutions is to estimate the state, $\mathbf{x}_t$, over time, as well as perhaps the parameters. The most basic solution to this problem is to use filtering techniques to consider the observation equation and the state equation at the same time. As a note, for convenience we will only consider the dynamical parameters, θ, and drop the the use of ϕ going forward.

2.2 Filtering Techniques

The filtering solution is foundational in that it not only solves the issue of filtering, but it is very closely related to the prediction problem, as well as the smoothing problem. The filtering method also plays a role in the estimation of model parameters for SSM and it will be the basis for the parameter estimation methods included in the `nLnG` package. The filtering solution requires that we determine the filter probability density, $p(\mathbf{x}_t|\mathbf{Y}_t, \theta)$. The filter probability density is defined in terms of the current state, $\mathbf{x}_t$, given all observations up to and including $\mathbf{Y}_t$, such that $\mathbf{Y}_t = \{\mathbf{y}_1', \mathbf{y}_2', \cdots, \mathbf{y}_t'\}'$. The solution for filtering is then broken down into a two step procedure: 1) model prediction and 2) measurement update. The technique can then be summarized into the general filtering algorithm for SSM as follows:

Algorithm 1 The general filtering algorithm for State Space Models

$p(\mathbf{x}_0|\mathbf{Y}_0, \theta) = p(\mathbf{x}_0)$
for t in 1 **to** N **do**
  Prediction Step
  $p(\mathbf{x}_t|\mathbf{Y}_{t-1}, \theta) = \int p(\mathbf{x}_t|\mathbf{x}_{t-1}, \theta)p(\mathbf{x}_{t-1}|\mathbf{Y}_{t-1}, \theta)d\mathbf{x}_{t-1}$
  Measurement Step
  $p(\mathbf{x}_t|\mathbf{Y}_t, \theta) \propto p(\mathbf{y}_t|\mathbf{x}_t, \theta)p(\mathbf{x}_t|\mathbf{Y}_{t-1}, \theta)$
end for

The filtering algorithm begins by setting the first filter density, $p(\mathbf{x}_0|\mathbf{Y}_0, \theta)$, equal to some initial state conditions, $p(\mathbf{x}_0)$, for the state, $\mathbf{x}_t$. The first filter density then acts as a starting point and is passed to the prediction step. In the prediction step the model is used to predict the state, $\mathbf{x}_{t-1}$, ahead in time and the predictive density, $p(\mathbf{x}_t|\mathbf{Y}_{t-1}, \theta)$, is calculated by integrating the model "transition" density, $p(\mathbf{x}_t|\mathbf{x}_{t-1}, \theta)$, and the filter density, $p(\mathbf{x}_{t-1}|\mathbf{Y}_{t-1}, \theta)$, at time, $t-1$, with respect to the state, $\mathbf{x}_{t-1}$. Statistical moments for the predictive state can then be calculated from the predictive density. In the measurement step a new observation, $\mathbf{y}_t$, becomes available and the prediction is refined by multiplying the predictive density by the likelihood of the observation, $p(\mathbf{y}_t|\mathbf{x}_t, \theta)$, and that is proportional to the new filter density, $p(\mathbf{x}_t|\mathbf{Y}_t, \theta)$, at time, t (following Bayes' Theorem). Statistical moments for the filter state can then be calculated from the filter density.

By using the above algorithm, the probability density function (pdf) for $\mathbf{x}_t$ can be determined using the information from the model predictions and observation updates. As briefly discussed in section 1.1, there are a number of techniques for filtering depending on the type of problem. There will be two filtering techniques included as tools in this new package. The Kalman filter is the first of the two and it is used for SSMs with linear systems and Gaussian noise structures. Since the Kalman filter does not work for NLNG SSMs, a sample based version of the filter, known as the particle filter, is used instead to handle these types of problems and it

is the second of the filters included in the package.

2.2.1 The Kalman Filter

The Kalman filter is used to estimate the true system state from a set of noisy observation measurements made about the state for a linear/Gaussian system. The system model and the set of observations are used recursively to predict and update the state and produce an optimal estimate of the true state (Kalman, 1960). Consider the case for the general SSM from section 2.1. The Kalman filter requires that the error structures for both the state, $\mathbf{v}_t$, and observation, $\mathbf{e}_t$, equations be normally distributed with a mean of zero. The state and observations noise must be specified as $\mathbf{v}_t \sim N(0, \mathbf{Q}_t)$ and $\mathbf{e}_t \sim N(0, \mathbf{R}_t)$. Since $\mathbf{v}_t$ and $\mathbf{e}_t$ are both normal random vectors, the system state $\mathbf{x}_t$ will also be a normal random vector. First let us state the linear/Gaussian system:

$$\mathbf{x}_t = \mathbf{D}_t \mathbf{x}_{t-1} + \mathbf{v}_t, \tag{2.3}$$

$$\mathbf{y}_t = \mathbf{H}_t \mathbf{x}_t + \mathbf{e}_t, \tag{2.4}$$

for use in the text below. The Kalman filter hence assumes that the values $d(\cdot)$ and $h(\cdot)$ (Eqn. 2.1 and 2.2) are linear and are represented by the matrices, $\mathbf{D}_t$ and $\mathbf{H}_t$, in the above equations.

The Kalman filter algorithm follows the general filtering algorithm above in Algorithm 1. The written algorithm for the Kalman filter is as follows:

- Setup initial conditions $\hat{\mathbf{x}}_{0|0} \sim N(\mu_{x_0}, \mathbf{P}_0)$
- For $t = 1$ to $N - 1$

  Prediction Step:

  * compute predictive mean: $\hat{\mathbf{x}}_{t|t-1} \leftarrow \mathbf{D}_t \hat{\mathbf{x}}_{t-1|t-1}$
  * compute prediction covariance: $\mathbf{M}_t \leftarrow \mathbf{D}_t \mathbf{P}_{t-1} \mathbf{D}_t' + \mathbf{Q}_t$

  Measurement Step:

  * compute filter covariance: $\mathbf{P}_t \leftarrow (\mathbf{M}_t^{-1} + \mathbf{H}_t' \mathbf{R}_t^{-1} \mathbf{H}_t)^{-1}$
  * compute Kalman gain factor: $\mathbf{K}_t = \mathbf{P}_t \mathbf{H}_t' \mathbf{R}_t^{-1}$
  * compute filter mean: $\hat{\mathbf{x}}_{t|t} \leftarrow \hat{\mathbf{x}}_{t|t-1} + \mathbf{K}_t(\mathbf{y}_t - \mathbf{H}_t \hat{\mathbf{x}}_{t|t-1})$

- End For

The initial state conditions, $p(\mathbf{x}_0)$, need to be normally distributed with mean, μ_{x_0}, and covariance, $\mathbf{P}_0$, and have to be supplied as a starting point for the filter in $\hat{\mathbf{x}}_{0|0}$. The Kalman filter computes a mean and covariance matrix of the system state, $\mathbf{x}_t$, at each step of the process. During the prediction step, the predictive mean, $\hat{\mathbf{x}}_{t|t-1}$, is obtained by running the model forward in time by applying the dynamics matrix, $\mathbf{D}_t$, to the previous filter state, $\hat{\mathbf{x}}_{t-1|t-1}$. The forecast error covariance matrix, $\mathbf{M}_t$, is calculated by adding the previous filter covariance, $\mathbf{P}_{t-1}$, with the state error covariance matrix, $\mathbf{Q}_t$. In the measurement step, the new filter covariance is computed by taking the inverse of the prediction error covariance, $\mathbf{M}_t$, adding the observation error covariance, $\mathbf{R}_t$, related to the process by the observation operator matrix, $\mathbf{H}_t$ (Equation 2.4) and taking the inverse of it. The filter covariance is then used to calculate the Kalman gain function, $\mathbf{K}_t$ (Equation 2.2). The new filter mean, $\hat{\mathbf{x}}_{t|t}$, is computed by adding the prediction mean and a correction term, which is the Kalman gain function multiplied by the difference between the observations, $\mathbf{y}_t$, and the prediction mean with the observation operator matrix applied to it. The Kalman gain function acts as a weighting factor for the correction term,

$$\mathbf{K}_t = \mathbf{P}_t \mathbf{H}_t' \mathbf{R}_t^{-1}. \tag{2.5}$$

The gain function is formed by combining the error covariance matrix, denoted as $\mathbf{P}_t$, and the observation operator matrix, $\mathbf{H}_t$, with the inverse of the observation noise error structure, $\mathbf{R}_t$. The state estimates of $\mathbf{x}_t$ is now conditioned on the observations up to and including time, t.

The implemented algorithm for the Kalman filter is presented in section 3.4 and a synthetic data set produced from an example animal tracking problem is also analysed to show its functionalities. Let us now look at the other filtering technique, the particle filter and show how its design is capable of handling NLNG types of SSMs.

2.2.2 The Particle Filter

The particle filter's main goal is much the same as it was with the Kalman filter. We wish to estimate the true state of the system, except now for models not only of the linear Gaussian SSM class (Equations 2.3 and 2.4), but the NLNG SSM class as well (Equation 2.1 and 2.2). To do that the particle filter needs a way of getting at the time evolution of the pdf $p(\mathbf{x}_t|\mathbf{Y}_t, \theta)$ that does not involve the updating of a mean and variance since the SSMs are no longer of the linear Gaussian form. To solve this problem the solution was to use a set of samples, also called ensembles or 'particles', to represent a distribution for the filter density. The sample has the following relationship:

$$\{\mathbf{x}_{t|t}^{(i)}, w_t^{(i)}\} \sim p(\mathbf{x}_t|\mathbf{Y}_t, \theta), \tag{2.6}$$

where $\{\cdot\}$ represents the collection of all samples or particles members of the state, such that $\mathbf{x}_{t|t}^{(i)}, w_t^{(i)}$ is the ith member of the weighted sample drawn from $p(\mathbf{x}_t|\mathbf{Y}_t, \theta)$, and "$\sim$" refers to "is a draw from".

The sample represents the distribution of the filter density at time t. It is made up of particles of the state, $\mathbf{x}_{t|t}^{(i)}$, and the weights, $w_t^{(i)}$. This generated sample set is used by the prediction and the measurement steps of the filter algorithm to approximate the joint distribution of the system state at the given time t. With the estimated state distribution obtained at each time-step, relevant statistics can be calculated for the

distribution if desired. This approach to filtering is very useful, as this technique can provide exact solutions to NLNG SSM problems in the limit of very large sample sizes. Note that there has been more than one method proposed for particle filtering. The method we will focus on is the most common one based on using a SMC algorithm. There is an excellent guide that illustrates many of the SMC concepts and theory behind them by Doucet and Johansen (2011). I would encourage any reader with an interest in SMC or MCMC methods to follow up there, as it presents the topics in an easy to understand and well organized manner.

The earliest versions of the particle filter used a sequential importance sampling (SIS) algorithm (Geweke, 1989), but SIS had a major flaw as weights could collapse down to a single particle in the set (meaning that all weights are zero except for a single particle), leading to what's called *particle degeneracy*. Generally SIS just does not work in practise. The solution to this problem was offered by introducing a re-sampling stage in the measurement update part of the SIS algorithm. Thus the most common method of particle filtering arose, Sequential Importance Re-sampling (SIR). This particle filter method will be an important inclusion in the new package of tools being created. The SIR filter functions as the fundamental process in the various state and parameter estimation methods available in `nLnG` and we will look at those in an upcoming section. Let us now introduce the SIR method in further detail.

2.2.3 Sequential Importance Resampling (SIR)

The SIR algorithm for the particle filter has had a number of versions over time. The Gordon et al. (1993) version of the algorithm was the first to introduce the updated recursive Monte Carlo algorithm of SIR, where they showed its superiority over a modified Kalman filter, the extended Kalman filter (EKF), when working with NLNG SSMs. In the measurement update step of the filter algorithm, Gordon et al. (1993) introduced the idea of re-sampling with replacement, which is accomplished

by re-sampling $\{\mathbf{x}_{t|t-1}^{(i)}\}$ with the probability proportional to the weight at time t, $w_t^{(i)}$, or by what is called a weighted bootstrap. This allows the set of particles to contain more of the most probable particles in comparison to the new observation made available.

The version of the SIR algorithm being used in package `nLnG` will be from Ionides et al. (2006). Ionides et al. (2006) version of the algorithm is very similar to the version by Gordon et al. (1993), except that the re-sampling stage is performed using a systematic updating version. The written algorithm is presented in the following manner using the notation established in Chapter 2. The SIR algorithm follows the general filter algorithm, Algorithm 1 presented in Section 2.2.

For each t over time span N:

1. Suppose recursively that $\{\mathbf{x}_{t-1|t-1}^{(i)}\}$ has (approximately) a marginal density of $p(\mathbf{x}_{t-1}|\mathbf{Y}_{t-1}, \theta)$
2. Prediction: Make $\mathbf{x}_{t|t-1}^{(i)}$ a draw from $p(\mathbf{x}_t|\mathbf{x}_{t-1} = \mathbf{x}_{t-1|t-1}^{(i)}, \theta)$. Then $\{\mathbf{x}_{t|t-1}^{(i)}\}$ has (approximately) the marginal density of $p(\mathbf{x}_t|\mathbf{Y}_{t-1}, \theta)$
3. Measurement: Now draw $\mathbf{x}_{t|t}^{(i)}$ from $\{\mathbf{x}_{t|t-1}^{(i)}\}$ with probabilities proportional to the re-sampling weights $w_t^{(i)} = p(\mathbf{y}_t|\mathbf{x}_t = \mathbf{x}_{t|t-1}^{(i)}, \theta)$. $\{\mathbf{x}_{t|t}^{(i)}\}$ has (approximately) a marginal density of $p(\mathbf{x}_t|\mathbf{Y}_t, \theta)$,

The SIR algorithm can still face the problem that the number of unique particles in the sample can diminish over time (or what is known as *sample impoverishment*). One solution to this problem is to use a larger particle set, the trade-off being that the Monte Carlo error is slightly greater for computed values and computation time tends to be greater. The use of SMC methods aims at lessening the loss of unique particles in the sample sets. A common procedure after a filtering solution is obtained for a SSM is to run the filtered results through a smoothing process. This is done since the filter uses only the past observations and for many studies the complete data set is available for state estimation. This is described below.

2.3 Smoothing Techniques

The smoothing solution that we will use for SSMs is called the forward-backward smoothing algorithm by Rauch (1963). It requires that a forward sweep through the observations is made by a filtering process first. The smoothing process is performed using a backwards sweep through time estimating the state distribution at time, t, considering all the observations up to some other later point in time, N. The smoothing solution depends on having the conditional likelihood for the observations:

$$p(\mathbf{Y}_N|\mathbf{x}_0, \mathbf{x}_1, \cdots, \mathbf{x}_N, \theta) = \prod_{t=1}^{N} p(\mathbf{y}_t|\mathbf{x}_t, \theta) = \prod_{t=1}^{N} p(\mathbf{e}_t), \tag{2.7}$$

where $\mathbf{Y}_N$=$(\mathbf{y}'_1, \mathbf{y}'_2, \cdots, \mathbf{y}'_N)'$. The joint density function of the states is:

$$p(\mathbf{x}_0, \mathbf{x}_1, \cdots, \mathbf{x}_N|\theta) = p(\mathbf{x}_0|\theta) \prod_{t=1}^{N} p(\mathbf{x}_t|\mathbf{x}_{t-1}, \theta) = p(\mathbf{x}_0|\theta) \prod_{t=1}^{N} p(\mathbf{v}_t). \tag{2.8}$$

These pieces are used to make up the conditional density function for the smoothed posterior distribution in the following manner:

$$p(\mathbf{x}_0, \mathbf{x}_1, \cdots, \mathbf{x}_N|\mathbf{Y}_N, \theta) = \frac{p(\mathbf{Y}_N|\mathbf{x}_0, \mathbf{x}_1, \cdots, \mathbf{x}_N, \theta) p(\mathbf{x}_0, \mathbf{x}_1, \cdots, \mathbf{x}_N|\theta)}{p(\mathbf{Y}_N|\theta)}. \tag{2.9}$$

This can be rewritten with the above definitions as:

$$p(\mathbf{x}_0, \mathbf{x}_1, \cdots, \mathbf{x}_N|\mathbf{Y}_N, \theta) \propto p(\mathbf{x}_0|\theta) \prod_{t=1}^{N} p(\mathbf{x}_t|\mathbf{x}_{t-1}, \theta) \prod_{t=1}^{N} p(\mathbf{y}_t|\mathbf{x}_t, \theta) \tag{2.10}$$

$$\propto p(\mathbf{x}_0|\theta) \prod_{t=1}^{N} p(\mathbf{v}_t) \prod_{t=1}^{N} p(\mathbf{e}_t). \tag{2.11}$$

This gives us the probability density of the states, $\mathbf{x}_0, \mathbf{x}_1, \cdots, \mathbf{x}_N$, conditioned on the set of observations, $\mathbf{Y}_N$, and the parameter(s), θ. This is different from the filtering result, where only one new observation, $\mathbf{y}_t$, is available at each measurement step. The smoother solution considers all the observations simultaneously. For this reason the

smoothing solution for SSMs often produces a smoother prediction of the true state than the filter produces for an estimate. This is a result of having future information available (the observation states) that is relative to the time of analysis, which allows for inferences to be made about the system noise (Rauch, 1963). This feature alone distinguishes the smoothing solution from the filtering solution for SSMs. The general forward-backward smoothing algorithm has the following form:

Algorithm 2 The general forward-backward smoothing algorithm

Require: forward filter sweep to collect $p(\mathbf{x}_N|\mathbf{Y}_N, \theta)$

Setup initial conditions $p(\mathbf{x}_N|\theta)$ at the final t, $t = N$

for t in $N-1$ **to** 0 **do**

$p(\mathbf{x}_t|\mathbf{Y}_N, \theta) = p(\mathbf{x}_t|\mathbf{x}_{t-1}, \theta)p(\mathbf{y}_t|\mathbf{x}_t, \theta)$

end for

The new package `nLnG` will include solution methods for both linear Gaussian and NLNG SSMs for the smoothing problem. We will examine each of these smoothing algorithms next.

2.3.1 Kalman Smoother

The Kalman smoother algorithm being using in `nLnG` was first introduced by Rauch (1963). The Kalman smoother is akin to the Kalman filter in that it requires that the SSM to be from the linear and Gaussian class of SSM. The system of equations produced can then be turned into the following written algorithm for the Kalman smoother with the following form:

1. Setup and run the Kalman filter. Store $\hat{\mathbf{x}}_{t|t}$, $\mathbf{P}_t$ and $\mathbf{M}_t$ for $t = 0, \cdots, N$.
2. Starting with the filter estimates at the final time $\hat{\mathbf{x}}_{N|N}$, $\mathbf{P}_N = \mathbf{P}_{N|N}$. Run the algorithm backward in time from $t = N - 1$ to $t = 0$.
   (a) Compute an updated gain matrix $\mathbf{K}_t^* = \mathbf{P}_t\mathbf{D}'_{t+1}\mathbf{P}_{t+1|t}$

(b) Compute the smoother variance $\mathbf{P}_{t|N} = \mathbf{P}_t + \mathbf{K}_t^*(\mathbf{P}_{t+1|N} - \mathbf{M}_{t+1})\mathbf{K}_t^{*'}$

(c) Compute the smoother mean state $\hat{\mathbf{x}}_{t|N} = \hat{\mathbf{x}}_{t|t} + \mathbf{K}_t^*(\hat{\mathbf{x}}_{t+1|N} - \mathbf{D}_{t+1}\hat{\mathbf{x}}_{t|t})$

The Kalman smoother follows the forward-backward smoothing algorithm represented in Algorithm 2. After a forward sweep by the Kalman filter, the smoother is run backwards in time using the the final positions of the filter estimates as initial conditions. For each time-step calculating the updating the Kalman gain matrix, $\mathbf{K}_t^*$, (Equation 2.12), the smoother error covariance, $\mathbf{P}_{t|N}$, and the smoother mean, $\hat{\mathbf{x}}_{t|N}$.

$$\mathbf{K}_t^* = \mathbf{P}_t\mathbf{D}'_{t+1}\mathbf{P}_{t+1|t}. \tag{2.12}$$

The Kalman smoother included in nLnG is fully demonstrated with the sample problem in Chapter 3.6. The Kalman smoother though, like the filter, lacks the ability to handle models in the NLNG framework. There are modified versions such as the extended, and ensemble Kalman smoothers which aim at solving this problem. Where the Kalman smoother is generally thought of as being a computationally efficient algorithm, the NLNG smoothing algorithms are much more difficult and computationally heavy. The NLNG smoother choice for inclusion in the new package is an SMC based method, the particle smoother, and is discussed next.

2.3.2 Particle Smoother

The particle smoother application for nLnG is a main focus for the development of this package. Since no packages for R currently include an algorithm for the particle smoother, nLnG will be the first to do so. The particle smoother algorithm replaces the updating of means and variance with the computation of a set of samples, particles or ensembles. This set of samples represents the joint distributions for the system state using all available observational information. The smoothing algorithm requires that on the forward sweep through the observations (running the particle filter) the samples, $\{\mathbf{x}_{t|t-1}^{(i)}\}$, of the joint distributions, $p(\mathbf{x}_t|\mathbf{Y}_{t-1}, \theta)$, be stored for each timestep

t. The algorithm being used for this inclusion of the particle smoother is from Godsill et al. (2004). This algorithm again uses the recursive forward-backward algorithm presented in Algorithm 2, though it uses an SMC version. The algorithm uses an SIR method to calculate the weights, $\mathbf{w}_{t|t+1}^{(i)}$, for sampling the smoother state, $\tilde{\mathbf{x}}_t$. The written version of the algorithm has the following representation:

- Choose $\tilde{\mathbf{x}}_N = \mathbf{x}_N^{(i)}$ with probability $\mathbf{w}_N^{(i)}$
- For $t = N - 1$ to 1
  - Calculate $\mathbf{w}_{t|t+1}^{(i)} \propto \mathbf{w}_t^{(i)} p(\tilde{\mathbf{x}}_{t+1}|\mathbf{x}_{t|t}^{(i)}, \theta)$ for each $i = 1, \cdots, j$
  - Choose $\tilde{\mathbf{x}}_t = \mathbf{x}_t^{(i)}$ with probability $w_{t|t+1}^{(i)}$ when $w_{t|t+1}^{(i)} > \alpha$
- $\tilde{\mathbf{x}}_{1:N} = (\tilde{\mathbf{x}}_1, \tilde{\mathbf{x}}_2, \cdots, \tilde{\mathbf{x}}_N)$ is an approximate realization from $p(\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_N|\mathbf{Y}_N, \theta)$

The particle smoother algorithm starts with selecting one of the saved state particles, $\mathbf{x}_{N|N}^{(i)}$, from running the filtering process (in this case the particle filter) as the starting condition for the smoother, $\tilde{\mathbf{x}}_N$. The smoother then runs backwards in time starting at $N - 1$ calculating the weights, $\mathbf{w}_{t|t+1}^{(i)}$. The weights are defined by the following equation:

$$\mathbf{w}_{t|t+1}^{(i)} = \frac{\mathbf{w}_t^{(i)} p(\mathbf{x}_{t+1}|\mathbf{x}_{t|t}^{(i)}, \theta)}{\sum_{j=1}^{N} \mathbf{w}_t^{(j)} p(\mathbf{x}_{t+1}|\mathbf{x}_{t|t}^{(j)}, \theta)}. \tag{2.13}$$

In Equation 2.13 the weight, $\mathbf{w}_t^{(i)}$, attached to particle $\mathbf{x}_t^{(i)}$ is applied to the state transition density $p(\mathbf{x}_{t+1}|\mathbf{x}_t^{(i)}, \theta)$ and is then normalized by the collected sum of the state transition densities over j. The weights, $\mathbf{w}_{t|t+1}^{(i)}$, are then used to sample the smoother state at time, $\tilde{\mathbf{x}}_t$, when that weight's probability is higher than the randomly generated uniform distributed value, $\alpha \sim U(0, 1)$. Once the algorithm runs backwards in time until $t = 0$, the sample $\tilde{\mathbf{x}}_{1:N}$ will be an approximate sample from the posterior distribution, $p(\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_N|\mathbf{Y}_N, \theta)$. One of these samples is called a realization and generally you wish to generate many of these realizations, as each of these realizations are independent from each other. The realizations of the conditional distributions,

$p(\mathbf{x}_0, \mathbf{x}_1, \cdots, \mathbf{x}_N | \mathbf{y}_{1:N}, \theta)$, are then used to approximate the smoothing distributions so that relevant summary statistics can be computed.

The implemented algorithm for the particle smoother (Godsill et al., 2004) is presented in Chapter 3.7. An example of its functionality using the synthetic animal tracking data is demonstrated as well. The downfall of the particle smoother is that when computing large numbers of realizations for large particle sets, the computation cost becomes very large very quickly. The particle set for the smoother needs to be the same size as the set used for the filter, since the smoother uses the saved filter densities, $p(\mathbf{x}_t | \mathbf{Y}_t, \theta)$, to update the smoother posterior density. Generally running larger number of particles for the filtering is a good idea, though the smoother only samples one particle from the set to be the approximation at that time t. It is possible to keep computational time reasonable for the smoother by keeping the particle set size smaller for the filter, at the cost of possible *sample impoverishment* during the filtering process. We have now introduced the filtering and smoothing methods that are included in the new package, let us move to the last area of tools we wish to cover for SSMs. SSMs generally require the ability to estimate model parameters, and we consider this problem next.

2.4 Parameter Estimation

The estimation of parameters for the SSM is usually one of the main tasks when examining a data set. A data set is usually a set of observed states of some phenomenon, which need a state equation, or process model, identified for them and a observation equation or model that describes the act of measurement. These models need to have parameters that account for effects on the phenomenon of interest or in our case, animal behaviours. SSMs require estimation of either the dynamics parameters or the statistical parameters from Eqns 2.1 and 2.2. The dynamics parameters needed for the state and observation models are comprised of the dynamic parameter,

θ, the observation operator parameter, ϕ, and the initial condition, $\mathbf{x}_0$. Other statistical parameters are comprised of the relevant quantities that describe the errors associated with each of the state and observation models, in this example case it will be $\mathbf{v}_t \sim N(0, \sigma_v^2)$, $\mathbf{e}_t \sim N(0, \sigma_e^2)$. There are various methods of estimation for SSMs, but we look only at the methods being included in the new package and detail how each work. We show how the methods can estimate both the dynamic and statistical parameters for the example SSM used Chapter 3. Maximum Likelihood Estimation (MLE) will be covered for both the linear Gaussian and NLNG SSM cases. Note: the MLE values returned in `nLnG` will not be an optimized solution, but will be in terms of the likelihood of the data given the parameter(s) for the test. Then we look at the two SMC methods for estimation based on using the particle filter as an 'engine' for the procedure. Many of the estimation methods being included in the new package are also covered by Wong (2012) and the reader may find it a good reference for the methods presented here. First let us introduce the likelihood based methods that both the filtering procedures in `nLnG` use to obtain estimates of parameters.

2.4.1 Maximum Likelihood Estimation (MLE)

The methods for MLE are based on the filtering solutions presented in Section 2.2. MLE's goal is to determine which parameter values are most consistent with the data and those values will maximize the likelihood function. The likelihood function for the two filters (Kalman filter and particle filter) are equivalent in the general formation as shown below:

$$L(\theta|\mathbf{Y}_N) = p(\mathbf{Y}_N|\theta) = \prod_{t=1}^{N} p(\mathbf{y}_t|\mathbf{Y}_{t-1}, \theta), \tag{2.14}$$

where θ is the vector of unknown parameters. From this point the methods start to differ. We will start by looking at the method implemented in the Kalman filter. It uses the concept of innovations to construct the likelihood function. Innovations are

defined as follows:

$$\mathbf{v}_t = \mathbf{y}_t - \hat{\mathbf{y}}_{t|t-1}, \tag{2.15}$$

where $\hat{\mathbf{y}}_{t|t-1} = \mathbf{H}_t\hat{\mathbf{x}}_{t|t-1}$ is referred to as the one-step ahead prediction error. We also need the innovations' error covariance matrix. It is defined as follows:

$$\mathbf{F}_t = \mathbf{H}_t\mathbf{M}_t\mathbf{H}_t' + \mathbf{R}_t. \tag{2.16}$$

So $\mathbf{v}_t \sim N(0, \mathbf{F}_t)$ and if $p(\mathbf{v}_t)$ is the density function of the innovations, the likelihood function for the Kalman filter can be written as follows:

$$L(\theta|\mathbf{Y}_N) = \prod_{t=1}^{N} p(\mathbf{v}_t|\theta). \tag{2.17}$$

The log-likelihood function is then constructed as:

$$logL(\theta|\mathbf{Y}_N) = constant - \frac{1}{2}\sum_{t=1}^{N} log|\mathbf{F}_t| - \frac{1}{2}\sum_{t=1}^{N} \mathbf{v}_t'\mathbf{F}_t^{-1}\mathbf{v}_t. \tag{2.18}$$

We convert the likelihood function into the log-likelihood function for convenience, since working with the logged version of the function is generally much easier. During the Kalman filtering process innovations and the error covariance matrices are calculated for each timestep, t. They are then used in the log-likelihood function to calculate the likelihood of the parameters.

Let us look at how the likelihood function changes when using the particle filter (SIR) for calculating the MLE in NLNG problems. Instead of calculating innovations, the sample of particles and the observed data is used for constructing the likelihood. From the general likelihood function Equation 2.14 above, $p(\mathbf{y}_t|\mathbf{Y}_{t-1}, \theta)$ is now the conditional density of the observations, and can be calculated as:

$$p(\mathbf{y}_t|\mathbf{Y}_{t-1}, \theta) = \int p(\mathbf{y}_t|\mathbf{x}_t, \theta, \mathbf{Y}_{t-1})p(\mathbf{x}_t|\mathbf{Y}_{t-1}, \theta)d\mathbf{x}_t. \tag{2.19}$$

Now $p(\mathbf{y}_t|\mathbf{x}_t, \theta, \mathbf{Y}_{t-1}) = p(\mathbf{y}_t|\mathbf{x}_t, \theta)$ and can be calculated since the observation density is based on the density of observation error, $\mathbf{e}_t$, which is known. $p(\mathbf{x}_t|\mathbf{Y}_{t-1}, \theta)$ is the

predictive density and we can drawn samples from it, $\{\mathbf{x}_{t|t-1}^{(j)}\} \sim p(\mathbf{x}_t|\mathbf{Y}_{t-1},\theta)$ of size J. We can now approximate the right-hand side of Equation 2.19 as follows:

$$\int p(\mathbf{y}_t|\mathbf{x}_t,\theta,\mathbf{Y}_{t-1})p(\mathbf{x}_t|\mathbf{Y}_{t-1},\theta)d\mathbf{x}_t \approx \frac{1}{J}\sum_{i=1}^{J} p(\mathbf{y}_t|\mathbf{x}_{t|t-1}^{(i)},\theta). \quad (2.20)$$

Then by substituting Equation 2.20 into 2.14, we get the following likelihood function:

$$L(\theta|\mathbf{Y}_N) \approx \prod_{i=1}^{N}\frac{1}{J}\sum_{i=1}^{J} p(\mathbf{y}_t|\mathbf{x}_{t|t-1}^{(i)},\theta), \quad (2.21)$$

and the log-likelihood takes the form:

$$logL(\theta|\mathbf{Y}_N) \approx \sum_{t=1}^{N} log(\sum_{i=1}^{J} p(\mathbf{y}_t|\mathbf{x}_{t|t-1}^{(i)},\theta)) - NlogJ. \quad (2.22)$$

So during each timestep t of the filtering run, the MLE is calculated by summing the weights, $\mathbf{w}_t^{(i)}$, from the SIR phase of the measurement update and then summing them together to get a approximation of the log-likelihood for parameter estimation.

The log-likelihood methods for parameter estimation in the new package `nLnG` will be illustrated in Chapter 3.8 for the synthetic animal tracking data set and when examining the real observed data in Chapter 4 to estimate the parameters for the unknown SSM. We will now move to estimation methods for parameters that are based on running the particle filter.

2.4.2 State Augmentation

Another method that uses the particle filter for parameter estimation is state augmentation. The goal of this method is to estimate the joint density function of the state and the unknown parameter(s). This allows for the parameter vector, θ_t, to evolve through time and provides an estimate of the parameter at that timestep (Kitagawa, 1998). The idea of this method is to augment the state vector to include the unknown

parameter(s) in the following manner:

$$\dot{\mathbf{x}}_t = \begin{bmatrix} \mathbf{x}_t \\ \theta_t \end{bmatrix}. \tag{2.23}$$

The parameter vector now evolves as $\theta_t = \theta_{t-1} + \mathbf{e}_t^\theta$, where the moments of $\mathbf{e}_t^\theta$ must be specified. We can now write the augmented states in the following SSM form as:

$$\dot{\mathbf{x}}_t \quad = f(\dot{\mathbf{x}}_{t-1}) + \dot{\mathbf{e}}_t, \tag{2.24}$$

$$\mathbf{y}_t \quad = g(\dot{\mathbf{x}}_t) + \mathbf{e}_t. \tag{2.25}$$

The error term is augmented as well to include the parameter error, as well as the state error as follows:

$$\dot{\mathbf{e}}_t = \begin{bmatrix} \mathbf{v}_t \\ \mathbf{e}_t^\theta \end{bmatrix}. \tag{2.26}$$

Now that we have the parameter(s) included in the augmented state vector, $\dot{\mathbf{x}}_t$, we can use the particle filter (SIR) algorithm to obtain the joint filter density for the state vector. Note that the state evolution is now nonlinear, and therefore the particle filter must be used. The sample or particle set now has this relationship:

$$\{\dot{\mathbf{x}}_{t|t}^{(i)}\} = \{\mathbf{x}_{t|t}^{(i)}, \theta_t^{(i)}\} \sim p(\mathbf{x}_t, \theta_t | \mathbf{Y}_t). \tag{2.27}$$

Although parameter estimation by state augmentation has the upper hand over MLE by having far better computational times. MLE provides estimation for static parameters while state augmentation estimates a time varying parameter and results are not really consistent with MLE results. The problem with state augmentation is that parameter values will not converge to static values. If the parameter(s) values do not evolve through time (i.e. $\theta_t = \theta_{t-1}$, then the filtering will "collapse". The solution to this problem is switching to using either multiple iterative filtering (covered next) or another method like particle MCMC. An example of state augmentation is presented in Chapter 3.9 with the synthetic animal tracking data set to illustrate this problem.

2.4.3 Multiple Iterative Filtering (MIF)

The last of the parameter estimation tools to be included in the new package is multiple iterative filtering (MIF). It is important to include a method in the package where estimates of parameter values will converge to static values. MIF is one such method (Ionides et al., 2006). MIF also uses the particle filter as the engine for the procedure. The method uses the idea of state augmentation, but includes a modification in that it decreases the variance of the parameter's error term over a specified number of runs of the state augmented particle filter (Ionides et al., 2006; Wong, 2012). The goal of MIF is to use parameters evolving through time to produce optimal estimates of time invariant parameters. By decreasing the variance of the error term, the parameter value will converge to values that approximately represent their corresponding MLE values as the variance of $\mathbf{e}_t^\theta$ gets closer to zero (Ionides et al., 2006; Wong, 2012).

The algorithm used in package `nLnG` for the `mif` procedure was introduced by Ionides et al. (2006) (a newer version of the algorithm was introduced in Ionides et al. (2011)). The goal of the algorithm is to maximize the likelihood in terms of the parameters, θ. The value that maximizes the likelihood is then the most appropriate value to represent that parameter in the SSM. The written version of Ionides et al. (2006) algorithm has the following form:

1. Select starting values $\hat{\theta}^{(1)}$, variance for starting values σ, a discount factor $0 < \alpha < 1$, an initial variance multiplier c^2, and the number of iterations T.

2. For n in $1, \cdots, M$

   (a) Set $\sigma_n = \sigma\alpha^{n-1}$. For $t = 1, \cdots, N$, evaluate $\hat{\theta}_t^{(n)} = \hat{\theta}_t(\hat{\theta}^{(n)}, \sigma_n)$ and $V_{t,n} = V_t(\hat{\theta}^{(n)}, \sigma_n)$ using the state augmented particle filter.

   (b) Set $\hat{\theta}^{(n+1)} = \hat{\theta}^{(n)} + V_{1,n} \sum_{t=1}^{N} V_{t,n}^{-1}(\hat{\theta}_t^{(n)} - \hat{\theta}_{t-1}^{(n)})$, where $\hat{\theta}_0^{(n)} = \hat{\theta}^{(n)}$.

3. Take $\hat{\theta}^{(M+1)}$ to be the maximum likelihood estimate of the parameter θ for the fixed parameter model.

The algorithm begins in step one with setting the all the initial conditions for the use of the MIF. The starting points for the parameter(s) evolving are selected, $\hat{\theta}^{(1)}$, the initial variance multiplier, c^2 (that inflates the variance of the parameters error term in the beginning iterations of the MIF), the discount factor, α, (that controls the level of decrease in the variance of the parameters error term on each iteration of the filter) and the number of iterations for the filter to make M. In step two the filter is iterated for 1 to N, decreasing the starting point of the variance, σ_n, by the discount factor, α, to the power of the iteration number minus one. The state augmented particle filter is then run over the time vector, $t = 1, \cdots, N$, calculating the following equations:

$$\hat{\theta}_t^{(n)} = \hat{\theta}_t(\hat{\theta}^{(n)}, \sigma_n) = E[\theta_t|\mathbf{Y}_t]. \tag{2.28}$$

$$V_{t,n} = V_t(\hat{\theta}^{(n)}, \sigma_n) = Var(\theta_t|\mathbf{Y}_{t-1}). \tag{2.29}$$

The algorithm uses the normal distribution with the moments shown above to update the random walk and as σ decreases towards zero we can get an estimate of θ. At the end of each of the particle filter iterations the parameter estimate, $\hat{\theta}^{(n+1)}$ is updated by the parameter estimation at time n added to the sum of the different of parameter estimate at time t and $t-1$ weighted by the variance at time t. In step three at the end of the MIF iterations the parameter estimate $\hat{\theta}^{(M+1)}$ can be considered to be the MLE of the parameter, θ.

The one real downfall with the MIF method is that it is computationally heavy as it runs the augmented particle filter (SIR) many times to get MLE estimates of the parameter values, but it is likely more efficient computationally than MLE. The main issue with particle filter MLE is that the likelihood is subject to Monte Carlo variation from it being computed from samples. The implemented algorithm for MIF is presented in Chapter 3.10 and an illustration of the functions use with the synthetic animal tracking dataset is shown. Now that we have introduced the background for

the applications to be included in nLnG. Package creation in R, the finer details of package nLnG's creation will be covered next. We will also look at the implemented version of the algorithms for each of the tools in nLnG and see their use with the synthetic animal tracking dataset.

Chapter 3

An R Package: nLnG

During this chapter the details of package creation in R are discussed as well as the package developed for this thesis: nLnG. The tools available in the nLnG package will be presented in detail for the reader. For each tool that is included in nLnG, we will present the implemented pseudocode for each of the included algorithms and also illustrate the use of the package throughout the chapter using a synthetic animal tracking data set. Also, each of the parameter estimation tools in the package will be demonstrated with the synthetic dataset, and a short discussion is included on how the results for each method compare. Appendix C at the end of the thesis is included to verify that the two filters and smoothers in the package produce matching results when the SSM is a linear Gaussian system.

The synthetic dataset we will use to introduce the tools in the nLnG package will mimic simple animal tracking data. The state vector $\mathbf{x}_t$ will be equivalent to Equation 1.1, describing the longitude and latitude position of the animal. The state equation and the observation equation for the problem are of the general form from Section 2.1, though the dynamics, $d(\cdot)$, and relation, $h(\cdot)$, functions are both only identity matrices with the dynamics/relation parameter on the diagonals rather than having more complex structure. The synthetic model will also include a drift term, δ. The state equation has the following form:

$$\mathbf{x}_t = \delta + \mathbf{x}_{t-1} + \mathbf{v}_t \tag{3.1}$$

The observation equation is then described as:

$$\mathbf{y}_t = \mathbf{x}_t + \mathbf{e}_t \tag{3.2}$$

```
theta  <-c(d=1.00,h=1.00,drift=0.05,Xlon.0=-60.5,
            Xlat.0=44.0,slon=0.15,slat=0.075,sOb=0.25)
```

Listing 3.1: Parameter vector to be used for the example tracking problem.

Table 3.1 specifies the parameters for the synthetic tracking model. The comparison showing the link between the mathematical symbols and the variables supplied to the model in a named vector as shown in Listing 3.1 above. The values for the identity dynamics and relation parameters are given as `d=1.00` and `h=1.00`, as are the standard errors for the longitude, `slon=0.15`, the latitude, `slat=0.075` and for the observation model, `sOb=0.25`. There is also a drift parameter included in the vector; it is set as `drift=0.05` for each coordinate. The initial state positions for the longitude/latitude coordinates are set to -60.5/44.0 degrees respectfully, with `Xlon` and `Xlat` tagged with `.0` at the end. Now that we described the setup of the problem, the reader can continue on to see how we encode the problem into the `nLnG` object in Section 3.2. However, we will start with covering the topic of package creation in R and giving some of the details of the creation of package `nLnG`.

Mathematical Symbol	`nLnG` Variable
$\mathbf{x}_{t,long}$	`Xlon.0`
$\mathbf{x}_{t,lat}$	`Xlat.0`
θ	`d`
ϕ	`h`
$\mathbf{v}_{t,long}$	`slon`
$\mathbf{v}_{t,lat}$	`slat`
$\mathbf{e}_t$	`sOb`
δ	`drift`

Table 3.1: This table shows the linking between the mathematical symbol in the state and observations equations and the variables supplied in `nLnG`

3.1 Package Creation

3.1.1 Introduction

The procedure for building packages for the statistical programming language R is well documented on the CRAN website in their "Writing R Extensions" guide (R Core Team, 2014). R packages necessarily consist of functions implementing the relevant methods, sample data and examples that users can run, as well as users guides. Historically one would go about putting the folder for the package together manually, but packages have become so common within R that the developers have included a function `package.skeleton` in the methods package that will prepare the base structure of a package for the user. There is an excellent guide for using the `package.skeleton` function by Leisch (2009) "Creating R Packages: A Tutorial". The function requires that the user supply a package name and the R source code base for the package. It then returns a source folder for the package (named by the package name the user supplied in your current workspace) containing the R code, a base structure for all the manual (`man`) pages for the functions, a base `description` file, a base `NAMESPACE` file, and a `read-then-delete-me` file with instructions on how to add compiled code libraries to the package. R is compatible with a number of programming languages to use as compiled code libraries from **C**, **C++**, etc. The compiled code is placed in a new folder called `src` inside the package folder created by `package.skeleton`. Compiled code generally offers an improvement in calculation times over what would be required by the base R program. The compiled code has the advantage that once complied it has memory allocated to it in advance, instead of at run time or at initialization of the function. Some additional files and folders are needed depending on which programming language is used for the compiled code. See the extensions guide for more details.

3.1.2 Linking Complied Code

Compiled code can be linked to R through a number of different methods, even by using other packages. We will only give details of the methods used for writing extensions for R mentioned earlier in the chapter. The methods contain details for **C**, **C++** and **FORTRAN**. Each of these languages can have compiled code embedded into R, using interface functions `.C` or `.Fortran`. The interface function contains the name of the function from the compiled code (which will be a `void` function) and the data mappings from the R atomic vector nodes to the type for each argument of the function, meaning the function assigns to each variable the type of data it will be (i.e. integer, double, etc). For passing R objects to compiled code the user can use either the `.Call` or `.External` function by supplying the function name for the complied code and the lists of variables needed to be passed to the function from R (note the data type no longer needs to specified here). For working with R objects within compiled code the user can make use of the R Internals package. Instead of specifying the type of each argument now, the R Internals package lets the user use `SEXPs` (pointers) and it points the compiled code function to look at the spot that R has the object and/or variables saved in memory. The pointers come in different types called `SEXPTYPEs`, with each working to get different information types from the location in memory. For working with **C/C++** you call the R Internals package by including `Rinternals.h` in the header of your program file. One of the big advantages of using the R internals package is that the user can create, manipulate and return R objects in/from their **C/C++** code to your R code all by just using `SEXPs`. The use of R Internals can be somewhat rigorous as the amount of information in the functionality of it can be quite extensive. Other common R functions can be called within **C/C++** by including `R.h` in the header, while some of the special distribution functions can be called by including `Rmath.h`. The headers lets **C/C++** have access

to a set of internal functions in R (many of which are coded in **C**), which helps consistency and saves time.

3.1.3 Help Pages

The `man` pages for the package created by `package.skeleton` have only the base structure of the help file. These files contain only the arguments for the function and some sparsely filled in function uses, leaving the title, descriptions, examples, etc. for the user to define. R also offers a section on writing R Documentation files in their extension guide mentioned above to ensure problems can easily be searched for. The `.Rd` files are very similar to **LaTeX** (a document markup language/document preparation system) with a few exceptions mostly to do with how code is handled by the interpreter, since code segments are more common in `.Rd` files. The `package.skeleton` function will generate `man` files for every function, method and generic defined in the package; this can lead to a large set of help files for the user to write. It is best to combine as many of the common files together into one file, and then use the `Alias` command to link the keywords to be searched for that will trigger the help page to open. For example, when someone is searching for the way to define an attribute of a function, the function's `man` page is called instead of a `man` page just explaining the use of the attribute (since that would cause a lot of redundancy in the help files and saves programming time).

3.1.4 Package Management

The `NAMESPACE` file is a management system for the package. It controls which package variables are made available to be exported and what variables need to be imported from other packages. If the package has compiled code libraries, the top of the

`NAMESPACE` file will contain a call to load the libraries using the function `useDynLib`. The libraries are compiled when the package is loaded in R by calling either `library` or `require` in the session. The last of the files generated by using `package.skeleton` is the `description` file. It contains the basic package information fields such as version, title, date, description, license, author, maintainer, depends, and collate. Many of these are self explanatory, but an interesting one is the `depends` field which loads the required packages that the package needs to run. The other interesting field is the `collate` field; during installation its order determines the order in which the code is installed. This is important when the user has a class that might depend on another class being installed first, and collate lets the user set the order. The use of the `package.skeleton` function is very useful in getting the structure of a package prepared.

3.1.5 R-Studio

The next aid in package creation we will discuss is **RStudio**, which was created by JJ Allaire and his team and is a general working environment for writing code (or performing analysis) with R. This type of environment is generally referred to as an integrated development environment (IDE). **RStudio** acts as an interface with R to create a powerful working environment for users. It will not function without an active R installation. **RStudio** lets the user simultaneously see their workspace, a command line to run code, as well as multiple scripts from which code can be run directly and then another window for plots, files, and help files making it a very efficient environment for coding. **RStudio** lets the user create a project space for their R code, which includes the option to create a package, or a package with a **C++** library. When create a package is selected, it asks the user to supply the list of R source code for the package, which can be just browsed to, found and the directory added, or a

new directory can be created to hold the planned developed code. **RStudio** then acts much like the `package.skeleton` function to create a package folder in your current workspace with all the same file generation as `package.skeleton` does. The package can be installed, after some editing by the user to include the help files, fill in the `description` file and make any edits that the `NAMESPACE` file requires. A programmer that has experience with scripting languages can have the listed changes made by running a script and the package installed from inside **RStudio**, (if not the installation will fail without the changes being made first). I personally spent time trying to figure out now to get the installation not to fail before it was realized that a script was needed so that you could stay within the **RStudio** workspace to make testing the package easier. **RStudio** was very new at the time of the development of nLnG and the supporting documentation available was not as vast as it is currently. Other than a slight learning curve to begin with, **RStudio** is a great environment tool for either doing analysis or development with the R language.

3.1.6 Error Checking

Now that we have introduced the way in which the user can generate files and how they can organize the folder structure for the package they are creating, we will take a second to talk about checking the package for errors and installing it on a computer system. A common way to work in **UNIX** and **Windows** based systems, is to run commands from the command line or prompt. To run R from the command line for development, the user should have the following programs: LaTeX, The Inno Setup installer (the latest version), `Rtools` and the MinGW-w64 toolchain. Once the user has the listed programs installed, the last thing the user needs to do is to add the pathways for R (the user can add both the 32 and 64 bit bins) and `Rtools` to the **Windows** environment paths, so that the system can recognize the programs when

called from the command line. To perform a error test on the package, first navigate to the directory where the folder with the package files is located. A check on the folder can be performed to see if the package is installable by using the command `R CMD check -packagename`. This command prompts R to perform a number of checks on the package making the user aware of its correctness and generates a folder in the workspace named `-packagename.Rcheck`. If the package fails and is not installable, this folder will hold an `00install` file only. This file holds the details of the error information. If the package contains an `src` file for complied code to be attached, the user can have R check it for correctness by adding `--check-subdirs=yes` to the checking command above. Note that this also performs a check of the `man` page files. If the user has not filled in any details to the `man` pages yet, this will cause the package to fail the error check. The user can avoid this by just giving the titles for each of the `man` pages. R expects to be able to install the `man` (help) files for the package at the time of installation. Once the package is installable, the `-packagename.Rcheck` folder will be filled with the above report files on the error checking. It will also contain an R file with all the examples for the `man` files and a folder with the package name that contains the installable files. Note: when using **RStudio** for development, the same list of programs is needed as when working from the command line above. The working environment just automates the command line calls (user can also set the options for these under `Build->Configure Build Tools...`).

3.1.7 Installation

Now that the package can be installed, we can do this by running the command, `R CMD INSTALL -packagename`, from the command line and the package will be installed into the R libraries directory. By adding the suffix `--build` to the install command, R will then build the .tar file. Once the user can creates the .tar file for their package,

you can run `R CMD check --as-cran` on the file. This error checks the package so that it is ready to be uploaded as a package to CRAN. The user can now give their package to other users, or upload it to CRAN or a user community site like R forge. These same commands used from the command line can be used inside the **RStudio** environment to perform the checking, building and installing of the package as well. Now that we have looked at how to create the files, perform error checking on them and install the package to our computer, we can look at the different styles of programming for package creation in R.

3.1.8 Programming for Package Creation

Let us next introduce the two common object-oriented programming (OOP) system types within R. We need to first define what OOP classes and methods are, how they work and how they function differently in the two settings. When users perform analysis on data or models in R there is a good chance that they will want to perform multiple tests with that data, or those models. Creating classes is a way to group results together and save them in one space for both the data or models used, and this acts as the definition of an object in computer science. Methods in OOP act as functions that perform a specific calculation on a specific type of class and/or returns results that are stored in the class object. For example, calling methods on the object lets the user access parts of the attributes contained in the object for doing things such as summaries, plots, etc. When a package is loaded into R, its set of generic functions is loaded into R's table of available methods. Functions already in R's method table can be overloaded with other available methods if they have the same name as a method already in the table. To avoid conflicts, methods are looked up by name and the type of class object the method is designed for. Now that we have defined how classes and methods operate in packages, we can move on to defining the

R function	S3 definition
`class(x)`	Get or set the class attributes of x
`unclass(x)`	Remove class attributes of x
`methods(generic.function)`	All available S3 methods for a generic function
`methods(class="class ")`	Get all S3 methods for a particular class
`is(object)`	Return object's class and all super-classes
`is(object, class2)`	Tests if object is from class2
`str(object)`	Display the internal structure of an object

Table 3.2: The list of functions for working with S3 class and methods in R and the definition of what each function does

two class types associated with R.

Classes in R are either done in the **S3** or **S4** class types. The **S3** type class is equivalent to just adding a name to an object. The object is expected to have the correct information since there is no formal object requirements for the object to conform to. This makes the **S3** type of OOP much simpler to implement and gives users quite a bit of freedom in the way they wish to use it. The functions to generate the typical class and method objects for the **S3** type are summarized in Table 3.2. One function not mentioned in the table is `NextMethod`, which will implement a simple level of inheritance (a method of establishing the `Is a` relationship between objects, so that a class can pass attributes to child classes) between objects, given the user must be using a vector as the class object. **S3** classes are used in R for many things, though it is a rather loose system. For those looking for stricter guidelines, the choice should be the **S4** type of OOP system. The **S4** type of classes and methods conform much closer to that of other OOP languages.

To define a class object in **S4** the user uses `setClass`. The user defines a name for the object class, specifies if it contains any other class objects and lists a representation for the object attributes. The new object is created by a call to the `new` constructor function and is defined at the end of the function for the given class. Methods are set

R function	S4 definition
`setClass()`	Create a new class
`setMethod()`	Create a new method
`setGeneric()`	Create a new generic function
`new()`	Generate a new object for a given class
`getClass()`	Get the class definition
`getMethod()`	Get the method definition
`getSlots()`	Get the name and class of each slot
`@`	Get or replace the contents of a slot
`validObject()`	Test the validity of an object

Table 3.3: The list of functions for working with S4 class and methods in R and the definition of what each of the functions are for

using `setMethod` in **S4**. The user gives `setMethod` the function name the method is intended for, the signature of the object class the method is meant for, and the call to the function for the method to perform. The functions that the methods implement in the package can be made to belong only to the classes by using `setGeneric`. The user gives `setGeneric` the function's name, the function call to the method being set, and the keywords `standardGeneric` with the function name enclosed to have the function set as part of package. Users unfamiliar with **S4** will notice a difference in the function to return parts of the object. Many objects use the `$` to parse through the parts of the object, but in **S4** objects slots are accessed by using `@` instead. The rest of the functions for working with **S4** objects are summarized in Table 3.3, which gives the R functions for working with the **S4** class type along with what each of the function's definition is.

Looking at the two tables, Table 3.2 and 3.3, you can see how much more complex the **S4** class is to work with compared to the **S3** class. The **S4** classes have functions defined for getting the parts of the class or method, and for checking the validity of an object that **S3** classes do not have.

We will next look at some of the details that went into the creation of the nLnG package.

3.1.9 nLnG Package Creation Details

The package nLnG is designed for the R language, so that it can be made freely available to the statistical community. The package is built using the **S4** style of OOP system for a more formal design. Much of the basic design for nLnG is based on the R package pomp by King (2012), though the internal structures have been updated or edited to some degree for its use in nLnG. The main reason is that pomp does not contain linear Gaussian tools (i.e. Kalman filter and smoother) and nLnG has been adapted to feature these tools as well. Also, pomp has the added feature of having a solution method for the prediction problem for SSMs. However that problem is not considered important here, as the primary target is to develop the smoothing solution with nLnG. For this reason, while following the same programming principle, the class objects are quite different. For example, the pomp class object has a `skeleton` attribute for performing nonlinear forecasting, while the class object for nLnG has a smoothing density calculator attribute, `smooth.den`. A full comparison of the methods available for each package can be seen in Table 3.4. From the table you can see that pomp does not support any linear Gaussian methods, or smoothing methods, but does offer an additional parameter estimation method, particle MCMC.

The most notable internal structure of pomp that nLnG makes use of is the way user-supplied functions and their interface plug-ins are used for either discrete or continuous data. This was done to speed up the development time for nLnG, as normally, development time can be long and tedious. The user-supplied functions themselves are detailed in Section 3.2 and based on the versions used in pomp. The functions have all been edited to some extent for the incorporation of linear Gaussian methods.

Method	nLnG	pomp
particle filtering	Y	Y
Kalman filtering	Y	N
particle smoothing	Y	N
Kalman smoothing	Y	N
nonlinear forecasting	N	Y
MLE	Y	Y
mif	Y	Y
PMCMC	N	Y

Table 3.4: Method inclusion comparison for the 2 packages. Methods are listed on the left and are included/excluded in the package if indicated by "Y/N" in the column

The methods for the particle filter and MIF are based on methods available in pomp. The particle filter included in nLnG has been edited to calculate filter variances and particle quantiles, while the MIF has yet to be edited to incorporate these new quantities from the particle filter (See the future work section in the Conclusion 5.3 for further details on edits still to be made for nLnG). The rest of the methods, while written in the same style of coding practice as pomp, are all new developments for nLnG.

Having just described the methods that will be included in nLnG, let us now focus on the internal structure of the package. The representation of the class `nLnG` can be seen in the Appendix A. One will notice it contains many of the variables you would expect for SSMs. There are variables for data, time, states, parameters and name variables for the parts of the model. Note in particular the variables that hold functions for the class. These functions are used in some combination for each of the various methods of analysis in nLnG. A full description of which user-supplied functions are needed for which methods is in the following section. The functions are for either simulating the process or calculating the target densities. Only the `initializer` function does not have to be specified by the user. The user needs only

to include the initial conditions for the model in the parameter vector `parms` with a `.0` attached to them to indicate it is an initial condition. The object then parses the values and creates a default initializer. The variables that are defined as `nLnG.fun`-type functions take the user-supplied function and check if its a regular R function or if the user has chosen to include a piece of native code.

Native code is just code written in another programming language that is meant for decreasing the computation time for the assigned function. We will not demonstrate using native code in this thesis, users can refer to the advanced topic guide for pomp by King (2012) for a better description and guide on how to use this functionality.

Now that we have covered much of the `nLnG` object structure we can move on to the rest of the package. We can see that each of the major application methods included in the package returns an object class (see Listing A.2 for the `exportClasses` field in the `NAMESPACE` file) with the results for the application method encoded as parts of the class object. There have been many class methods designed for accessing the various parts of each of the returned class objects and many of these are illustrated later in this Chapter.

In the `NAMESPACE` file, we can see `useDynLib` attaches a large library of complied code that the package makes use of. This code is compiled upon installation and loaded into the environment when the package is called. Many of the NLNG applications in nLnG use complied libraries to help decrease the computational cost of these methods, as SMC methods are generally computationally heavy. (Note: in the `NAMESPACE` file the other packages from R that nLnG uses are all called by using the `importFrom` command in the file). The packages that nLnG depends on are mvtnorm, subplex and deSolve. These packages are loaded automatically when nLnG is called.

The `description` file for the package can be viewed in Listing A.3. The

`description` file is where the programmer includes the title, description of the package, information about his or her self, specifies what the package depends upon (including the version of R that is required), and sets the collate field for the order of installation. The license field is the type of license the package uses. It must be specified or the distribution of the package or the general use of the package may be illegal. nLnG makes use of the `GPL` license for versions greater than 2, meaning that its under the "GNU General Public License" holding for both version 2 and 3 of it. We have now covered the details involved in the creation of a package for R, as well as for the new package nLnG. Let us move on to demonstrating the use of nLnG by showing how to create an `nLnG` object.

3.2 Defining an `nLnG` State Space model object and Functions

The `nLnG` object requires that the user specifies the necessary parts of the object needed for running the tool they wish to use. Each of the tools in nLnG requires that a different subset of the function list below is encoded in the object. The below-listed parts of the `nLnG` object requires the user to encode the function that performs the specific action, such as advancing the state model from time $t-1$ to time t, being carried out by `state.sim`, or calculating the observation density for the particle filter, as `obsev.den` does. Depending on whether you wish to simulate data or analyse a set of observed data the user must specify a `data.frame` either with the named observed data and their times, or a `data.frame` with the named variables and the time specified for the data to be recorded from the simulation.

The list of functions in `nLnG` objects are:

1. state simulate function (`state.sim`) - advances the state model from time $t-1$ to t (Equation 3.1)

2. observation simulate function (`obsev.sim`) - converts the observation state to the process state or $\mathbf{y}_t$ to $\mathbf{x}_t$ (Equation 3.2)

3. state density function (`state.den`) - Included for future work (see note below)

4. observation density function (`obsev.den`) - computes the weights for the particle filter

5. state update function (`state.update`) - computes the predictive mean for the Kalman filter

6. smoother density function (`smoother.den`) - computers the weights the for particle smoother.

Each of the functions listed above will be introduced further in this chapter, as each function supports different procedures. The parts of the object needed for each of the procedures included in `nLnG` are summarized in the following list.

1. To simulate model, use functions 1 and 2

2. To perform Kalman filtering, use functions 1 and 5

3. To perform particle filtering, use functions 1 and 4

4. To perform Kalman smoothing, use functions 1 and 5

5. To perform particle smoothing, use functions 1,4 and 6

6. To perform multiple iterative filtering, use functions 1 and 4

The `state.den` function is currently not needed for any of the tools included in this release of the package, but was included for work that may be developed at a later date (see future work in Chapter 5.3). The requirements for all the tools include the

`state.sim` function. The `state.sim` function is the base for all the implemented tools, as each tool requires that the user can simulate from the state model.

The list of variables the user needs to supply for an `nLnG` object is as follows:

- `data.frame` with named variables
- `times` vector; may be included in `data.frame`
- `tInit` initial state time
- `parms` parameter vector
- Some combination of `state.sim`, `obsev.sim`, `state.den`, `obsev.den`, `state.update` and `smoother.den`
- optional initializer function to set the initial conditions
- optional covariance table `covar` and times for each covariate `tcovar`
- optional names vectors `obsnames`, `statenames`, `paramnames`, `covarnames` and `zeronames`

The `nLnG` object requires the user to specify the `times` variable. It can be specified, or a pointer to the times variable in the `data.frame` can be used, by `times="time"`. The user must specify the time for the initial conditions of the state `tInit`. The user must also specify the parameters for the model `parms` in a named vector. The initial conditions for the model are also included in the `parms` vector and are signalled by attaching a `.0` to the end of the variable name. The user then needs to supply the appropriate function for the tool they wish to use (described above). The user can optionally supply a function that sets the initial conditions of the model (but the `nLnG` object can create one from the `parms` vector; explained above). The user can optionally supply a covariate table for each time t by specifying `covar` and `tcovar`. The user can also optionally supply the names vectors for any of a number of variables

supplied (listed above; not required as long as the data is in named vectors). Now that we have detailed the parts of the `nLnG` object that needs to be specified by the user, let us see how to encode the example animal tracking SSM problem in package nLnG.

3.2.1 Example Problem: State Model Definition

We first need to write the `state.sim` and `obsev.sim` functions for producing the synthetic data. The `state.sim` function performs a single step in time, $t \to t + \Delta$, where Δ is the size of the time-step of the state model. The state model, Equation 3.1, for the synthetic tracking data is written as follows:

```
track.state.sim <- function(x,t,parms,delta.t,...){
  ##get dynamic parameter
  d <- parms["d"]
  drift <- parms["drift"]
  ##generate new states
  xnew <- rnorm(n=2,mean=c(drift,drift)*delta.t+
  d*x[c("Xlon","Xlat")],sd=(delta.t*parms[c("slon","slat")]))
  names(xnew) <- c("Xlon","Xlat")
  return(xnew)}
```

Listing 3.2: `state.sim` function

The function is straightforward. When called it gets the state vector, `x`, at time `t`, the `parms` vector containing the parameters (where you pull the appropriate parameter(s) out as seen above, Listing 3.2), and the time-step, `delta.t` (which is set to 1 by default). The function must return a named vector containing the updated state

vector at time, $t + \Delta$. Being able to set the variable `delta.t` to different values lets the user vary the time in which observations are made. This is because the measurements may not be recorded at the regular time intervals. For the synthetic data, the `state.sim` function in Listing 3.2 implements the RWM. The parameters values, `d` and `drift`, are extracted from the parameter vector. The new state vector, `xnew`, is then generated by using the `rnorm` function. The `mean` is set equal to the drift plus the previous state position, `x[c("Xlon","Xlat")]`, multiplied by the dynamic parameter, `d`, and the standard deviation, `sd`, is equal to the corresponding value for each of the longitude and latitude errors. The names of the variables are then added back to the new state vector, `xnew` (note the names returned in `xnew` must be the same as the names of the input vector `x`), and the updated states are returned. Now that we have defined the state function, we can move on to using the `simulate` function to produce data.

3.3 Creating an `nLnG` Object and Simulating Data (`simulate`)

While we will be simulating data for the synthetic animal tracking problem to illustrate the use of `nLnG`, in a general SSM problem, usually one starts with a set of noisy observations of the state of the system and not from simulated data. Simulating data from the model can be useful to verify the model is correct before proceeding to applications using the observed data. A general analysis will follow using `nLnG` with the real animal tracking dataset in Chapter 4.

The other function we need to define for simulating from the model is the observation model function `obsev.sim`. It models the measurement properties for the recorded observations. We define the function in the following manner:

```
track.obsev.sim <- function(x,t,parms,...){
 ##get relation parameter
 h <- parms["h"]
 ##get observed point and return
 y <- rnorm(n=2,mean=h*x[c("Xlon","Xlat")],sd=parms["sOb"])
 names(y) <- c("ylon","ylat")
 return(y) }
```

Listing 3.3: `obsev.sim` function

The function is again very straightforward. When called, it gets the state vector, `x`, at time, `t`, and the `parms` vector containing the parameters for the model. The function generates new observations for the state, then the observed state vector, `y`, is named, and returned. In the synthetic data `obsev.sim` function, Listing 3.3, the relation parameter, `h`, is extracted and then the observed states, `y`, are generated by using the `rnorm` function. The mean is set to the relation parameter multiplied by the current state vector, `x[c("Xlon","Xlat")]`, and the `sd` set to the observation error, `sOb`. Using the `names` commands in R plays an important part in encoding the functions for the parts of the `nLnG` object. Using `names` lets the user add the labels for the data to the newly generated data, so that it is saved correctly in the object. Now that we have the functions specified for simulating the synthetic data, let us generate the observed animal track that we will be working with for the rest of this chapter.

The first thing we need to do to simulate data is to create the object for the model. Creating a `nLnG` object is done by calling the self-named `nLnG` function. The user needs to specify a number of the attributes for the object and assign it to a variable inside the R workspace. The variable will then be of class `nLnG` and hold all of the attributes of the class. The R object created will hold the data and the user defined

functions for performing the required procedure (Listing 3.2 and 3.3 in this example case here). (Note: all the functions do not need to be specified at one time for the object and they can be added as needed. We will show this as we add the functions to `track.ex` as we need them for each procedure throughout this chapter).

The object of class `nLnG` is created as shown in Listing 3.4 below. Listing 3.4 shows that the call to the `nLnG` function is saved in the variable, `track.ex`. Then the following list of `nLnG` object arguments are defined:

- `data` is set to a `data.frame`, with the `time` set from 1 to 50 and 2 empty observed variables holders (`ylon` and `ylat`)
- the `times` argument is pointed to the `data.frame`
- the initial state vector is at time zero, `tInit=0`,
- the model parameters are set to the parameters introduced earlier (Listing 3.1), `parms=theta`
- both the simulator functions (`track.state.sim` and `track.obsev.sim`) detailed in Section 3.2 and 3.3

It is important to note here that the `state.sim` variable has a function wrapper that needs to be supplied to describe the type of timestep it is taking. The three choices to use are `discrete.time.sim`, `onestep.sim` and `euler.sim`. Each stepping function represents a different method for advancing the state function. The `onestep.sim` wrapper is for when it is possible to simulate the state from one time to the next. The `euler.sim` wrapper is for when the state is advanced by a continuous number of smaller steps through time (method for using continuous-time distributions for the state function). The `discrete.time.sim` wrapper is for advancing the state from one time to another by as many timesteps as needed (used when time vector is discrete, but time-steps are uneven). For the example we will use the `discrete.time.sim`

wrapper and the reader can refer to the `man` pages for `state.sim` and `plugins` for more details on the rest of the wrappers. Using the `discrete.time.sim` wrapper requires that the user specify the stepping function `step.fun` (the state model in Listing 3.2) and the time-step between measurements, `delta.t`. In the example case here, `delta.t=1`, even though the time vector is of equal size step (this will be of more importance in Chapter 4 when the time vector is no longer even).

```
track.ex <- nLnG(data=data.frame(time=1:50,ylon=NA,ylat=NA),
                 times="time",tInit=0,
                 state.sim=discrete.time.sim(
                 step.fun=track.state.sim,delta.t=1),
                 obsev.sim=track.obsev.sim,
                 parms=theta)
```

Listing 3.4: `nLnG` object: Creating an object of class `nLnG` to hold the example animal tracking data

Now we have specified everything we need to simulate data from the model, all we have left to do is run the `simulate` function with the `nLnG` object `track.ex` and the parameters `theta`. For the purpose of this example we have set the seed of the random number generator (RNG) as `seed=1`. This is so that we are working with the same synthetic data for illustrating each of the tools in `nLnG`. The call to the `simulate` function can be seen in Listing 3.5 below.

```
> track.ex <- simulate(track.ex,parms=theta,seed=1)
```

Listing 3.5: Simulating data using the `simulate` function for the example tracking problem.

After calling the `simulate` function, the model will be simulated over the given time frame (1 through 50 in the case here) and the generated states are recorded in the `data.frame` under the data variable column inside `track.ex`. Now that we have the synthetic animal tracking data we should visualize it by plotting the data. This can be done by using the `plot` method included in `nLnG`. It produces one dimensional plots of all the data saved in the data frame of the object against the time and can be seen in the Appendix B. The `plot` function is meant to be a quick way of visualizing the data and does not support higher level plots. Since the tracking data is 2D, we will need to plot the longitude positions, `Xlon,Ylon`, against the latitude positions,

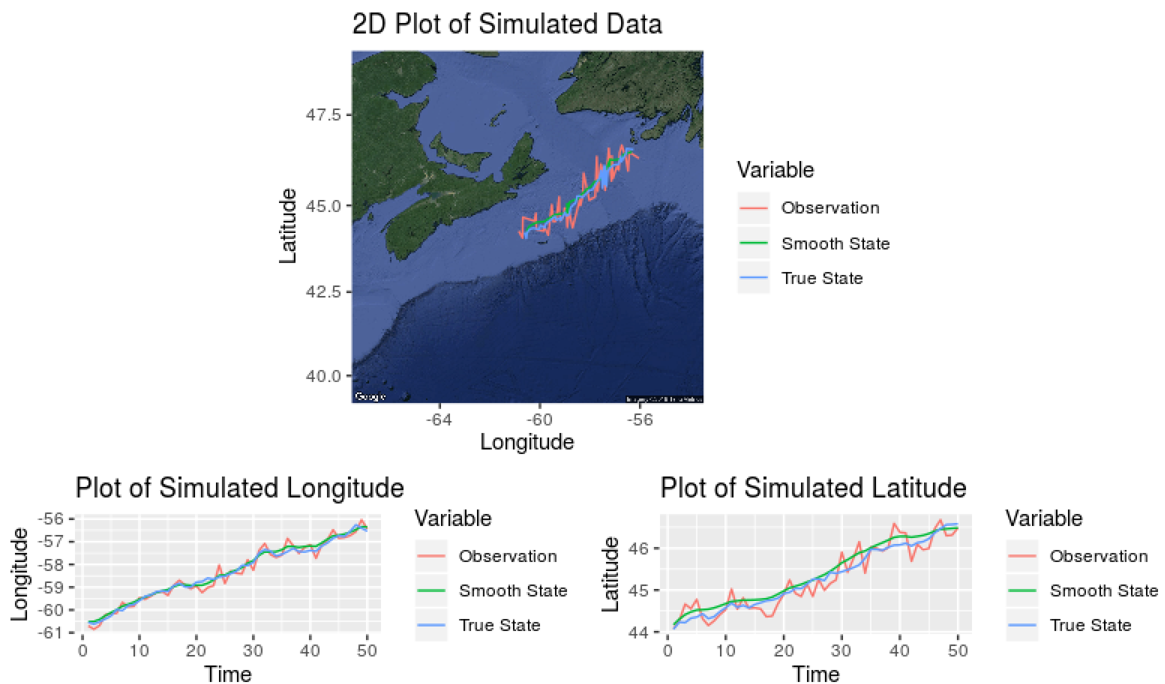


Figure 3.1: In the top plot of the figure, the simulated animal states (light blue) and observations (red) are plotted against each other for the 2D plot on a map of the Scotian Shelf. Underneath the 2D plot, each of the longitude and latitude state and observation variables are plotted individually against the timestep.

`Xlat,Ylat`, by a different method. The user can extract the `data.frame` from the `nLnG` object by using the `as.data.frame` method on the object.

After extracting the data and plotting it, Figure 3.1 shows that the true state (blue line) is fairly consistent throughout the plot, while the observed state (red line) is much less consistent (which makes since, the observation error was `sOb=0.25`). Now remember generally for an SSM problem we would only have the observed states to begin with. So that means really we would only have the observed states (red line) in Figure 3.1 and one can see how much they can deviate from the true states (light blue line). One of the main goals would be to eliminate the noise and recover the true location of the animal's position. To do this, the state and observation models would need to determine parameter estimates, but since we are using a test case SSM, it is possible to skip this for the time being and move right into the de-noising problem of the state estimation. We will first address the problem through the filtering solution methods in `nLnG`, starting with the Kalman filter.

3.4 The Kalman Filter (`kFilter`)

For the implementation of the Kalman filter (Kalman, 1960) for the package `nLnG` the written algorithm presented in Chapter 2.2.1 can be transferred to the following pseudocode in Algorithm 3.

The first thing in the algorithm is the command for the function `kFilter` and we can see that it requires that the user supplies an `nLnG` object and the parameters. Looking at the algorithm, the initial state, $N(\mathbf{x}_0, \mathbf{P}_0)$, is the starting point for the filter density, $\hat{\mathbf{x}}_{init}$. The filter is run over the length of the observation data (minus one) until $t = N$. In the prediction step, $\hat{\mathbf{x}}_{t|t-1}$ and $\mathbf{M}_t$ are updated. After they are updated a new observation becomes available, $\mathbf{y}_t$. The observation is then used to refine the prediction of the state in the measurement step. The new filter covariance, $\mathbf{P}_t$, is computed and used to determine the Kalman gain function, $\mathbf{K}_t$. The Kalman gain function acts as the correction term to update the new filter mean, $\hat{\mathbf{x}}_{t|t}$. The

algorithm then calculates the innovations, $\mathbf{v}_t$, and the innovation error covariance, $\mathbf{F}_t$. These are then used to calculate the conditional log likelihoods, $l_t(\theta)$, which are then summed for the log-likelihood estimate. Now that we have finished looking at the algorithm used for the implemented `kFilter` function, let us demonstrate its use with the example animal tracking data in `track.ex`.

Algorithm 3 `kFilter(object,parms,...)`

Require: nLnG object, parms

$\hat{\mathbf{x}}_{init} \leftarrow N(\mathbf{x}_0, \mathbf{P}_0)$

for t in 1 **to** $(N-1)$ **do**

$\hat{\mathbf{x}}_{t|t-1} \leftarrow \mathbf{D}_t\hat{\mathbf{x}}_{t-1|t-1}$

$\mathbf{M}_t \leftarrow \mathbf{D}_t\mathbf{P}_{t-1}\mathbf{D}_t' + \mathbf{Q}_t$

$\mathbf{y}_t \leftarrow yobs_t$

$\mathbf{P}_t \leftarrow (\mathbf{M}_t^{-1} + \mathbf{H}_t'\mathbf{R}_t^{-1}\mathbf{H}_t)^{-1}$

$\mathbf{K}_t \leftarrow \mathbf{P}_t\mathbf{H}_t'\mathbf{R}_t^{-1}$

$\hat{\mathbf{x}}_{t|t} \leftarrow \hat{\mathbf{x}}_{t|t-1} + \mathbf{K}_t(\mathbf{y}_t - \mathbf{H}_t\hat{\mathbf{x}}_{t|t-1})$

$\mathbf{v}_t = \mathbf{y}_t - \hat{\mathbf{x}}_{t|t-1}$

$\mathbf{F}_t = \mathbf{H}_t\mathbf{M}_t\mathbf{H}_t' + \mathbf{R}_t$

$l_t(\theta) \leftarrow log\det\mathbf{F}_t - \mathbf{v}_t'\mathbf{F}_t^{-1}\mathbf{v}_t$

end for

The `kFilter` function requires that the user specifies the `state.update` function for updating the mean in the prediction step of the algorithm. The `state.update` function acts very much like the `state.sim` function (from Section 3.2) and it could be considered more convenient for the user because they do not need to re-specify the `state.sim` function just for the `kFilter`, with the `sd` set to zero. This is needed because the forecast error covariance, $\mathbf{M}_t$, is updated separated from the prediction mean, $\hat{\mathbf{x}}_{t|t-1}$, as seen in Algorithm 3. Looking at the `state.update` function in Listing 3.6 for the example tracking problem, the user can see the function is very

much the same as the `state.sim` function in Listing 3.2. The function takes the same variables as the `state.sim` function does: the state vector, `x`, the time, `t`, the parameters, `parms`, the time-step, `delta.t` and anything else the user needs to specify is covered by the $\cdots$ in the input variables. In the code for `track.update`, the dynamics parameter is extracted from the `parms` and the position of the state at time t is saved in `x`. The update of the state/mean for `xnew` is still performed by using the `rnorm` function, the `mean` is set to Equation 3.1 minus the state error and the `sd` is set equal to zero. `xnew` is then relabelled with the variable names and returned to the `kFilter` function. Adding the function to the `track.ex` object is done in the same manner as adding the `state.sim` function to the object, by using one of the three stepping functions explained in Section 3.2. Adding `track.update` to the `track.ex` object, can be seen in Listing A.7. In Listing A.7, we use the `discrete.time.sim` stepping function to add the `state.update` function to the `track.ex` object with a `time.step` equal to one. Now that we have specified the parts of the `nLnG` object needed for running the `kFilter` function, we can look at how to use the command.

```
track.update <- function(x,t,parms,delta.t,...){
  ##get dynamics matrix
  d <- parms["d"]
  ##get position at x at time t
  x <- x[c("X1","X2")]
  ##update the mean
  xnew <- rnorm(n=2,mean=c(parms["drift"],parms["drift"])*
  delta.t+d*x,sd=0)
  names(xnew) <- c("X1","X2")
  return(xnew)}
```

Listing 3.6: `state.update` function for `kFilter`

The `kFilter` function requires the user to specify the `nLnG` object and the model parameters, `parms`, shown in Algorithm 3. The function also requires at the present time that the user supplies the state error covariance matrix, $\mathbf{Q}_t$, the observation error covariance matrix, $\mathbf{R}_t$, and the time-step, `delta.t`, (see future work in Chapter 5 for details on planned changes). The specified error matrices for the example tracking case can be seen in Listing A.7, with `sError` being the state error covariance matrix and `oError` being the observation error covariance matrix. With the error matrices specified we now have everything we need to call the `kFilter` command. The call to the `kFilter` function for the example tracking data can be seen in Listing 3.7. The `kFilter` function has been given the `track.ex` object with the encoded model, the parameter vector, `theta`, (shown in Listing 3.1), the two error matrices, `oError` and `sError`, and the time-step, `delta.t=1`. The bare minimum call to the `kFilter` will return an estimate for the log-likelihood for the parameters supplied to the function. The prediction and filter means and variances calculated are recorded for the procedure by turning on the logical statements for each, by setting it equal to `TRUE` (as seen in Listing 3.7). The output from the `kFilter` function is then saved in `track.kf`, which will be of class `kFilterd-nLnG`, for a Kalman filtered `nLnG` object.

```
track.kf <- kFilter(track.ex,parms=theta,obsev.error=oError,
                state.error=sError,delta.t=1,pred.mean=TRUE,
                pred.var=TRUE,filter.mean=TRUE,filter.var=TRUE)
```

Listing 3.7: Call to the `kFilter` function for the example problem `track.ex`, using the parameters, `theta`, and recording the results for the calculations.

The results saved in `track.kf` can then be extracted by using methods for the `kFilterd-nLnG` object. The log-likelihood estimate for the parameters can be extracted from the `kFilterd` object using the `logLik` method. Listing 3.8 shows the

call for `logLik` on the Kalman filtered example tracking object, `track.kf`, and that the log-likelihood estimate for the parameter vector, `theta`, is -26.38947 from the Kalman filter.

```
> logLik(track.kf)
[1] -26.38947
```

Listing 3.8: Extracting the Log-likelihood estimate from the object from the `kFilter` function for the example animal tracking problem using the `logLik` function in package nLnG

For further inspection of the Kalman filter results, the prediction and filter means and variances can be extracted using the `pred.mean`, `pred.var`, `filt.mean` and `filt.var` methods for the `kFilterd` object. Using these methods, we can use the R package ggplot2 (Wickham, 2016) (using the package ggmap (Kahle and Wickham, 2013) add-on for ggplot2 to obtain the map and create the map plot) to create the 2D plot in Figure 3.2 of the true states (the state from which the synthetic data were generated), observed states and the Kalman filter estimated states. Figure 3.2, shows that the filter state (red) follows much closer to the state (blue) than to the observations (green) and does a good job of estimating the true states. The filter mean does overreact to a number of the observations that are far from the actual state positions, which is expected with the size of the observation error, (`sOb=0.25`).

The user also has the option of using the built-in filter diagnostic function, `filt.diag`, to quickly visualize the filter results. In Listing A.8, the call to `filt.diag` for the example tracking problem can be seen. In the call, the user supplies the function with the `kFilterd` object, `track.kf`, gives the number of variables, two, gives the type of filter, `type="Kalman"`, and signals whether to produce a 2D plot or not, `Dimplot=TRUE`. The reader can refer to Figures B.2-B.5 for the diagnostics plots that the function produces. From the above and the Appendix B results, we can say that the filter did a good job of recovering the true values (as expected with

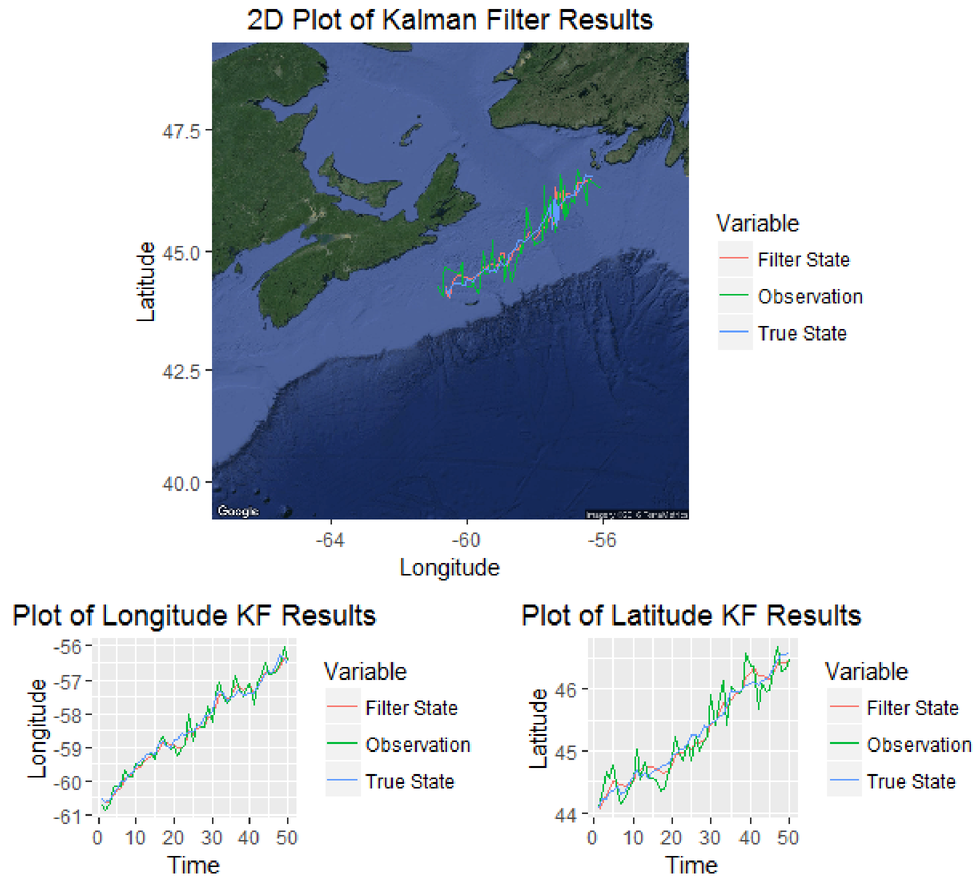


Figure 3.2: In the top plot in the figure, the filtered states (red), observations (green) and the true states (blue) are plotted against each other for the 2D plot on the map of the Scotian Shelf. Underneath the 2D plot, each of the longitude and latitude variables are plotting individually, with each having the filtered states (red), observations (green) and the true states (blue) plotted against the time

the example problem using synthetic data). Now that we have covered the Kalman filter will we next move to introducing the particle filter included in `nLnG`.

3.5 The Particle Filter (`pFilter`)

For the implementation of the particle filter for the package `nLnG`, the Ionides et al. (2006) algorithm presented in Chapter 2.2.3 can be transferred to the following pseudocode in Algorithm 4. Algorithm 4 first shows the command for the `pFilter` function. The user supplies the function with an `nLnG` object, the number of particles, `Np`, and the model parameters, `parms`. The pseudocode begins with showing that the algorithm requires that the same three user quantities just listed be specified and that `Np` must be greater than zero. The algorithm then sets up and creates an ensemble from the initial state, with the size of the ensemble equal to the specified number of particles, `Np`. The ensemble representing the state is then run through the filter over t from 1 to $(N-1)$. In the prediction step, the state particles are advanced forward in time by the transition density (the `state.sim` function acts as the transition density in `nLnG`). Next, in the measurement step, when the new observation, $\mathbf{y}_t$, becomes available the weights, $w_t^{(j)}$, are calculated (performed by the `obsev.den` function in `nLnG`). Within the complied function `pFilter_computations`, the weights are first normalized and used to compute the portion of log-likelihood estimate for the parameters by summing the weights and dividing by the particle size, J. The prediction mean and prediction variance are computed straight from the prediction density ensemble, $\{\mathbf{x}_{t|t-1}^{(j)}\}$. The filter mean and filter variance are calculated by applying the weights to the prediction density ensemble. The weights also are used in re-sampling the predictive density ensemble, by first being used to build a re-sampling index, I, by checking whether the cumulative weight value, c_p, is greater than a random uniform distribution between 0 and $w^{(j)}/Np$ and second by applying the I to the predictive density, $\{\mathbf{x}_{t|t-1}^{(j)}, w_t^{(j)}\}$. This creates a filter density ensemble, $\{\mathbf{x}_{t|t}^{(j)}\}$, of state particles that have high probability of being close to the observation. The conditional log-likelihood estimates, $l_t(\theta)$, are summed for a log-likelihood estimate,

$l(\theta)$, of the parameter θ (all of these computations are performed inside the compiled library `pFilter_computations` for `nLnG`).

Now that we have described the SIR algorithm being used for the particle filter in `nLnG`, let us illustrate how to use the function. We need to be able to calculate the target density for the distributions so we first need to write a observation density model, `obsev.den`, and add the function to the `nLnG` object, `track.ex`. The `obsev.den` function calculates the weights in the measurement step of the particle filter. The `obsev.den` function requires that the user supplies the observation position, `y`, the state vector, `x`, the time, `t`, model parameters, `parms`, and the `log` variable in the function call. The function is then specified by the user to calculate the density, $p(\mathbf{y}_t|\mathbf{x}_t = \mathbf{x}_{t|t-1}^{(j)}, \theta)$, of the observations, `y`, given the states, `x`. The function must contain a R distribution density function for determining the density of the function or it will not be accepted by the `nLnG` object. The function should return the calculated density as its output. The `obsev.den` function for the tracking example can be seen in Listing 3.9. The `obsev.den` function for the tracking example models the observation process for the equipment recording the animals position (or Equation 3.2 specified earlier). Looking at the user specified function for `obsev.den` in Listing 3.9, the operator parameter `h` is pulled out of the `parms` vector to begin with. In this example, `h` is just a identity matrix with the parameter on each of the diagonal entries. Next the incoming states (`x`), the observation (`y`) and observation error (`sOb`) are made into the following matrices, the observation, `Y`, the state, `xx`, and the observation covariance, `cv`. This is done because we have bivariate data and we wish to use `dmvnorm` to calculate the multivariate normal density, `fnew`. Finally, if the `log` variable is set to `TRUE`, the log-likelihood is returned, else the likelihood is returned in the value, `fnew`, from the `obsev.den` function.

Now that we have the `obsev.den` function specified, we can add the function to the `track.ex` object. Adding additional functions to the `nLnG` object is very easy and simple to do. The user simply re-creates the `nLnG` object with the arguments to

Algorithm 4 `pFilter(object,Np,parms,...)`

Require: an nLnG object, Np, parms.

Ensure: Np > 0.

$\mathbf{x}_0 \leftarrow \mathbf{x}_{init}$, $\theta_0 \leftarrow \theta_{init}$

$\{\mathbf{x}_{0|0}^{(Np)}\} \leftarrow \text{rep}(\mathbf{x}_0, Np)$

for t in $1:(N-1)$ **do**

$\{\mathbf{x}_{t|t-1}^{(j)}\} \leftarrow p(\mathbf{x}_t \mid \mathbf{x}_{t-1} = \{\mathbf{x}_{t-1|t-1}^{(j)}\}, \theta)$ (`state.sim`)

$w_t^{(j)} \leftarrow p(\mathbf{y}_t|\mathbf{x}_t = \mathbf{x}_{t|t-1}^{(j)}, \theta)$ (`obsev.den`)

$w_t^{(j)} \leftarrow w_t^{(j)} / \sum w_t^{(j)}$ (`pFilter_computations`)

$l_t(\theta) \leftarrow log(Np^{-1} \sum_{j=1}^{Np} w_t^{(j)}) = log\ p(\mathbf{y}_t|\mathbf{y}_{t-1}, \theta)$ (`pFilter_computations`)

$\hat{\mathbf{x}}_{t|t-1} = \frac{\sum_j \mathbf{x}_{t|t-1}^{(j)}}{J}$ (`pFilter_computations`)

$\hat{\sigma}_{\mathbf{x}_{t|t-1}} = \frac{\sum_j \mathbf{x}_{t|t-1}^{(j)} \mathbf{x}_{t|t-1}^{(j)} - (\sum_j \mathbf{x}_{t|t-1}^{(j)})/j}{J-1}$ (`pFilter_computations`)

$\hat{\mathbf{x}}_{t|t} = \frac{\sum_j \mathbf{x}_{t|t-1}^{(j)} w_t^{(j)}}{\sum_j w_t^{(j)}}$ (`pFilter_computations`)

$\hat{\sigma}_{\mathbf{x}_{t|t}} = \frac{\sum_j \mathbf{x}_{t|t}^{(j)} w_t^{(j)} \mathbf{x}_{t|t}^{(j)} - (\sum_j w_t^{(j)} * w_t^{(j)}) / \sum_j w_t^{(j)}}{\sum_j w_t^{(j)}}$ (`pFilter_computations`)

for j in $1:J$ **do**

while $U_j(0, (w_t^{(j)}/Np)) > c_p$, where $c_p = \sum_m w_m$ **do**

set $p = p + 1$, where $p_1 = 1$

end while

set $I_j = p$, where I is the re-sampled index

end for (`pFilter_computations`)

re-sample $\{\mathbf{x}_{t|t-1}^{(j)}, w_t^{(j)}\}$ according to I_j

$\{\mathbf{x}_{t|t}^{(j)}\} \sim p(\mathbf{x}_t|\mathbf{Y}_t, \theta)$

end for

$l(\theta) \leftarrow \sum_{t=1}^{N} l_t(\theta)$

---

the function being the previous `nLnG` object, plus any new arguments the user wishes to add to the object. The call to add the `obsev.den` function to the `track.ex` object

can be seen in Listing A.9. The first argument in the call is the `track.ex` object that was previously used and the second argument is adding `track.obsev.den` in Listing 3.9 to the object as the `obsev.den` function. After performing the call in Listing A.9, the `obsev.den` function is now part of `track.ex` object and we are ready to use the `pFilter` function in `nLnG`.

```
track.obsev.den <- function(y,x,t,parms,log,...){
  h <- parms["h"]
  xx <- matrix(c(x["Xlon"],x["Xlat"]),ncol=2,nrow=1)
  Y <- matrix(c(y["ylon"],y["ylat"]),ncol=2,nrow=1)
  cv <- matrix(c(parms["sOb"],0,0,parms["sOb"]),ncol=2
               ,nrow=2)
  ##get density
  fnew <- dmvnorm(Y,mean=xx,sigma=(cv%*%cv),log=log)
  return(fnew)}
```

Listing 3.9: `obsev.den` function

The `pFilter` function requires that the user supplies the `nLnG` object, the model parameters, `parms`, and the number of particles, `Np`, for its use. Running a `pFilter` function call with only the bare minimum requirements will result in it only returning a log-likelihood estimate for the model parameters supplied for the observed states, `y`. Measurement quantities can be turned on by logical statements in the call to the `pFilter` function for the predictive mean and variance, filter mean and variance and saved states for each time-step (which allows for quantile calculation). For the tracking example model, `track.ex`, the call to the `pFilter` function is in Listing 3.10. Looking at the call, the first argument is the `track.ex` object with all the functions and data stored in it, next the parameters are set to the vector `theta` (introduced at the beginning of this chapter) and the number of particles are set to `Np=1000`. The

additional arguments are all set to TRUE to tell the pFilter function that we wish to record the measurements and save them to the new pFilterd-nLnG object made when the pFilter completes. In Listing 3.10, all the findings from the filtering will be saved in track.pf. The new pFilterd-nLnG object is a subclass of the nLnG class, so it inherits all the attributes from the nLnG class and has its own class methods to access the different parts of the object. We can start looking at the results saved in track.pf for the filtering. The main methods of diagnostics are the calculated log-likelihood for the parameters and the effective sample size for the particles used with the filtering.

```
track.pf <- pFilter(track.ex,parms=theta,Np=1000,
                    pred.mean=TRUE,pred.var=TRUE,
                    filter.mean=TRUE,filter.var=TRUE,
                    save.states=TRUE)
```

Listing 3.10: Call to the pFilter function for the example tracking data, using the parameter vector, theta and recording the all the quantities from the calculations.

The log-likelihood for the track.ex object is obtained by using the class method logLik. The logLik method returns the summed conditional log-likelihoods (the individual conditional log-likelihoods can be accessed using track.pf@cond.logLik and its use is shown in Listing 3.11). The method is used by calling the logLik function on the filtered object track.pf. The estimated log-likelihood for the model parameters, theta, is -22.41174. This estimate of the log-likelihood is very close to the estimate seen in Listing 3.8 from the Kalman filter (-26.38947), with the difference between them being contributed to Monte Carlo error in the particle filter. In the MLE section, we will show how the estimate for the parameters values are obtained by profiling the likelihood for the value that maximizes the log likelihood.

```
> logLik(track.pf)
[1] -22.41174
```

Listing 3.11: Extracting the Log-likelihood estimate from the object from the `pFilter` function for the example animal tracking problem using the `logLik` function in package `nLnG`

The next thing we can do to examine the filtering results is look at the plot of the effective sample size. The effective sample size tells us how many of the particles were distinct during each step of the filtering. If the state or observations models are incorrect for the observed data, it is very likely the filtering will fail (almost all particle weights are near zero). The effective sample size can be plotted using the `effSampleTime` method included in `nLnG`. The call to the method's function can be seen in Listing A.4. The user gives the function the filtered object (in this case `track.pf`). The plot produced using `effSampleTime` can be seen in Figure 3.3. Referring to the plot, the number of effective particles is generally very high throughout and only drops below 200 effective particles once (Note: number of particles was 1000), around the 24th timestep. This is very good but since this is only a test problem and we know the true states, we should expect the effective sample size to be large. While these are good measures of performance for the modelling, we still might want to know more about the filter and predictive densities. The user can do this by turning on the calculations as shown in Listing 3.10 by setting them all equal to `TRUE`.

Turning on these logical attributes tells the `pFilter` function to record the calculations in the `pFilterd-nLnG` object. The predictive densities are represented by the prediction mean and variance at each timestep. They can be extracted from the object using the two `pFilter` methods `pred.mean` and `pred.var`. The methods return a matrix with the variables as rows and the timestep as columns. The filter densities also

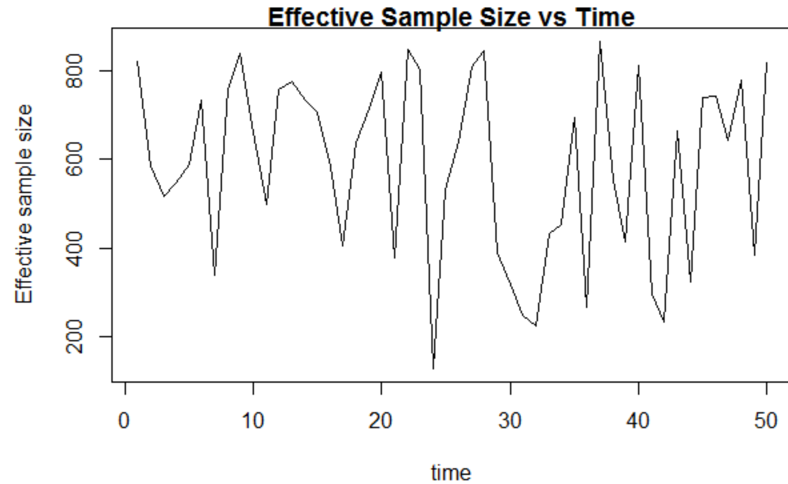


Figure 3.3: The plot of the effective sample size over time for the `track.pf`, where the total number of particles used was 1000

have a filter mean and variance recorded in the object and the methods for extracting each are `filter.mean` and `filter.var`. When the attribute `save.states=TRUE`, the filter densities for each time-step are saved and can be accessed in the `pFilterd` object by using `objectname@save.states`. The returned object from `save.states` is a `list`, with the time-step being the `list` number. These different measures generally are plotted to observe their behaviours. The user can extract the data from the `pFilterd` object (Listing A.5) and construct plots using one of the many plotting functions available in R.

For the example animal tacking data in `track.pf`, we have the advantage of knowing the true state, $\mathbf{x}_t$, so we can easily compare the results of the filter. We can create a plot like in Figure 3.4 of the `pFilter` results, by plotting the filter mean,

the observations and the true states. The plot shows that the filter mean (red line) is much closer to the true state (blue line) than to the observations (green line). The observation error for the example data is 0.25, so one would expect the filter state to trend more heavily towards the observations. But yet the `pFilter` result performs

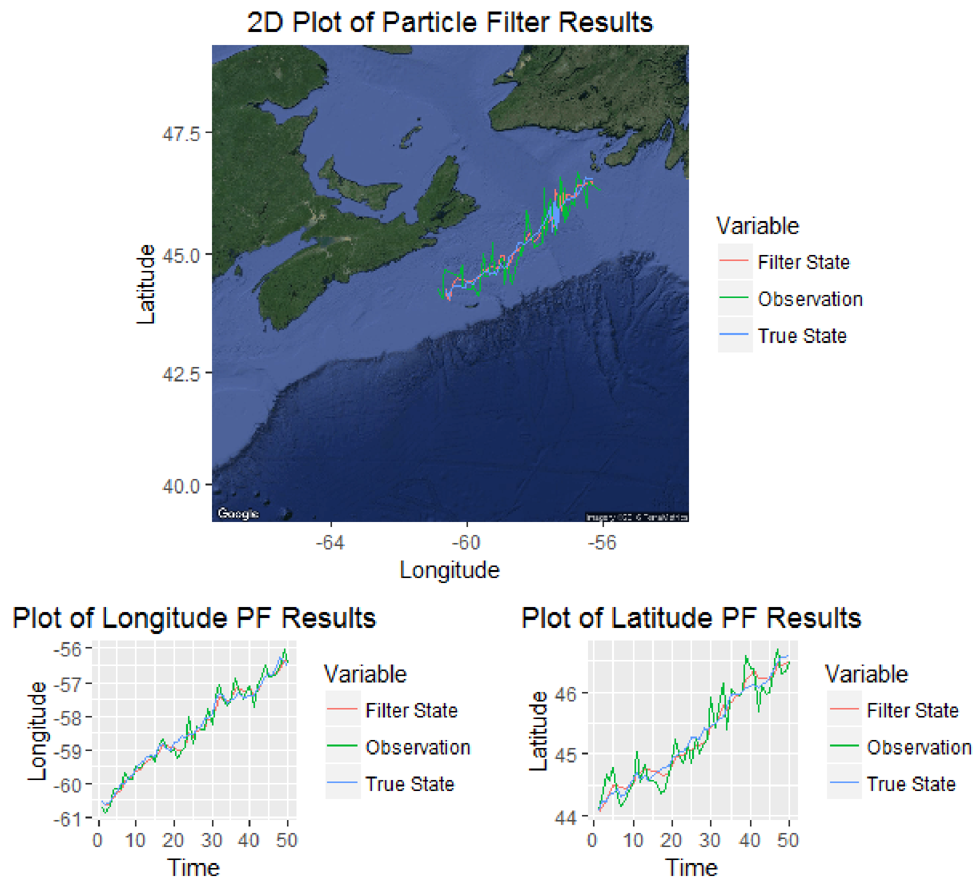


Figure 3.4: In the top plot in the figure, the filter states (red), observed states (green) and the true state (black) are plotted against each other to form the 2D plot from the particle filter results on the Scotian Shelf map. Underneath the 2D plot each of the longitude and latitude variables are plotting individually with the filter states (red), observed states (green) and the true state (black) being plotted against the time

much better with the filter mean being, while not a perfect match, a very good representation of the true state in this case. Since we had the true states for the

example problem we created our own plot, though `nLnG` does contain a plotting diagnostic function for quickly visualizing the filter results.

The user can call the `filt.diag` function on the `pFilterd` object produced from the filtering. The call to the function for the example problem can be seen in Listing A.6. The function gets the object, `track.pf`, the number of dimensions, `2`, type of filter, `="particle"`, and if a 2D plots should be created, `Dimplot=TRUE`. The plots that `filter.diag` produces are Figures B.6-B.10 in Appendix B. From the above and the appendix, it shows these results are fairly consistent with the results from the Kalman filter in the last section. For the comparison of the particle filter and Kalman filter results the reader can refer to the C.1 in Appendix C at the end of this chapter (showing the two procedures are equivalent for the linear Gaussian SSM problem).

Now we have introduced the algorithms for the two filters included in `nLnG` and seen them demonstrated with the example tracking data problem. We can now move on to the next procedure included in `nLnG`, which is smoothing. We will first look at the Kalman smoother.

3.6 The Kalman Smoother (`kSmoother`)

The Rauch (1963) algorithm, presented in Chapter 2.3.1, can be transferred to the pseudocode in Algorithm 5 for the implementation of the Kalman smoother in package `nLnG`. Algorithm 5 skips over the the first step of the written algorithm which is the Kalman filter, it is accomplished when the user runs the `nLnG` object through the `kFilter` function to create the `kFilterd-nLnG` object. This is effectively performing the forward sweep for step one of the written algorithm and an `kFilterd-nLnG` object is a part of the required information needed for the `kSmoother` function. The rest of the user required information needed for the function are the filtered states $(\hat{\mathbf{x}}_t, \mathbf{P}_t, \mathbf{M}_t)$ and the parameter set for the problem. The filtered states are saved in

the `kFilterd-nLnG` object by setting the prediction and filter means and variance logical to true (for an example see Listing 3.7). Algorithm 5 shows that the function call, `kSmoother()`, requires the user to specify the object with the saved information, the parameters, `parms`, for the model, and any other information for the function, $\cdots$ (this will be illustrated in the demonstration below).

The `kSmoother` function then performs the second step of the written algorithm. In Algorithm 5, it shows that the function first pulls the filtered states $(\hat{\mathbf{x}}_t, \mathbf{P}_t, \mathbf{M}_t)$ out of the object the user supplied. The procedure then runs backwards in time starting at the second to last step, $N-1$, and running till the initial time-step. During the iterating for each step three calculation are performed. First, a modified Kalman gain matrix is calculated by combining prediction variance and filter variance with the inverse of the dynamics parameters. This modified Kalman gain matrix is then used to weight the calculation of both the smoother variance, $\mathbf{P}_{t|N}$, and the smoother mean, $\hat{\mathbf{x}}_{t|N}$. Once the time-step reaches the initial time, the function completes and returned a new object of type `kSmootherd-nLnG`, with the saved results. Now that we have introduced the algorithm, we will illustrate the use of the function with the example tracking problem.

Algorithm 5 `kSmoother(kFilterd-nLnG object,parms,...)`

Require: `kFilterd-nLnG` object, filtered states$(\hat{\mathbf{x}}_t, \mathbf{P}, \mathbf{M})$, parameters

$\mathbf{P} \leftarrow$ `pred.var(object)`

$\mathbf{M} \leftarrow$ `filt.var(object)`

$\hat{\mathbf{x}}_t \leftarrow$ `filt.mean(object)`

for t in $N-1$ to 0 **do**

$\mathbf{K}_t^* = \mathbf{P}_t \mathbf{D}_{t+1}' \mathbf{M}_{t+1}$

$\mathbf{P}_{t|N} = \mathbf{P}_t + \mathbf{K}_t^* (\mathbf{P}_{t+1|N} - \mathbf{M}_{t+1}) \mathbf{K}_t^{*\prime}$

$\hat{\mathbf{x}}_{t|N} = \hat{\mathbf{x}}_t + \mathbf{K}_t^* (\hat{\mathbf{x}}_{t+1|N} - \mathbf{D}_{t+1} \hat{\mathbf{x}}_t)$

end for

The kSmoother function does not require the user to specify any additional transition or density functions for the procedure. The call to the kSmoother function for the example tracking data can be seen in Listing 3.12. It shows the function requires that the user supplies a kFilterd-nLnG object and the parameter vector, in this example case we have given the function the track.kf object from the Kalman filter result and the parameters vector, theta. We also have turned on the two logical attributes to record the smoother mean and variance with smoother.mean=TRUE and smoother.var=TRUE respectively. It also shows in Listing 3.12, that we have saved the results of the kSmoother function in the variable, track.ks, it will now be of class type kSmootherd-nLnG. Now that we have shown the reader how to use the kSmoother function, let us illustrate how to access and perform diagnostics on the results saved in the object created by the smoother function, track.ks.

```
track.ks <- kSmoother(track.kf,parms=theta,
                         smoother.mean=TRUE,
                         smoother.var=TRUE
                )
```

Listing 3.12: Call to the kSmoother function for the example tracking data, using the parameter vector, theta and recording the smoother mean and variance.

The functions smooth.mean and smooth.var are methods for the kSmootherd object class to extract the saved smoother mean and variance from the kSmootherd variable. In Listing A.10, it shows that the two functions are called on the object containing the Kalman smoother results, track.ks. By extracting the smoother mean, the 2D plot in Figure 3.5 can be created of the true state (blue), the observations state (red) and the smoother mean state (green). Figure 3.5 shows that the Kalman smoother mean state (green) again follows much closer the true state (blue) than to the observation state (red). This again shows that the estimated state follows the true

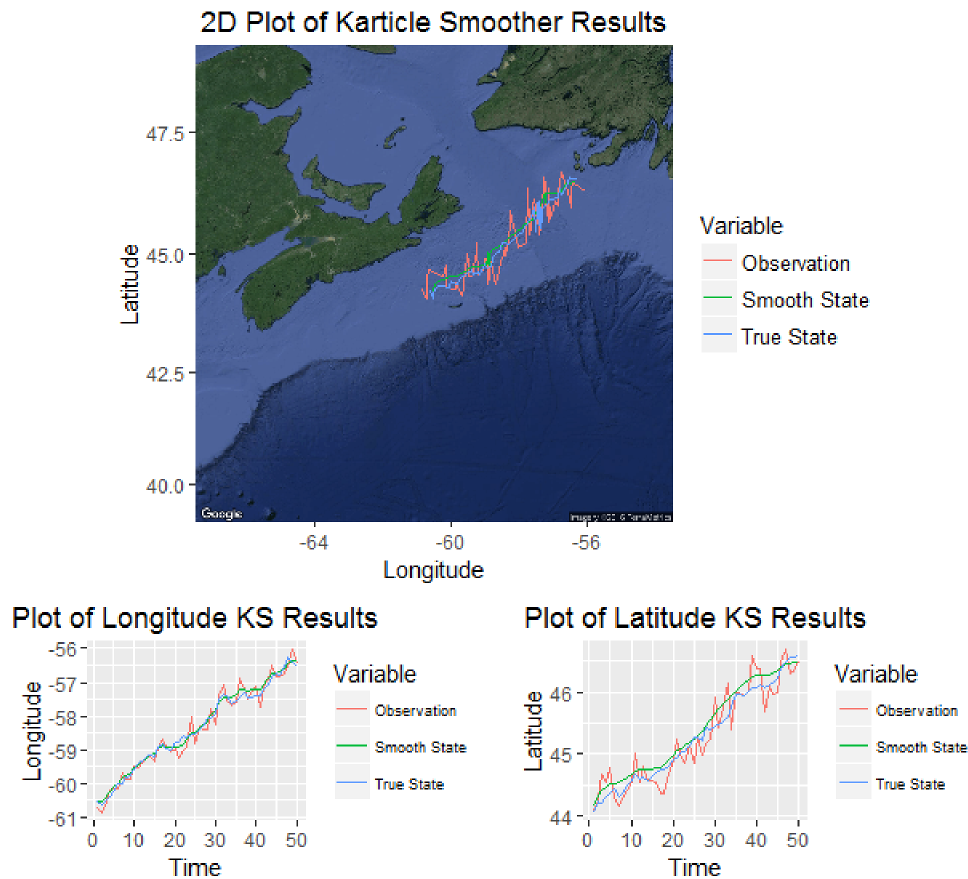


Figure 3.5: In the top plot in the figure, the smoother states (red), observed states (green) and the true state (blue) are plotted against each other to form the 2D plot from the Kalman smoother results on the Scotian Shelf map. Underneath the 2D plot each of the longitude and latitude variables are plotting individually with the smoother states (green), observed states (red) and the true state (blue) being plotted against the time

state of the animal much more closely than the observed location of the animal. It is safe to say that the procedure has done a good of recovering the true state/locations of the animal.

The `nLnG` package also contains the diagnostic function, `smoother.diag`, that allows the user to be able to quickly visualize the results of a smoothing test. For the example tracking problem, the user call for `smoother.diag` can be seen in Listing

A.11. From the above results and the appendix findings, we can say the results of the `Ksmoother` application are consistent with what we were expecting. The smoother estimate state tends to be through the middle of the filter states and the estimates of the variance are very close to the estimates that were returned from the Kalman filter.

Now that we have detailed the `kSmoother` function for package `nLnG` and illustrated its use with the example tracking problem for the user. We can move on to the second smoother included in `nLnG`, the particle smoother.

3.7 The Particle Smoother (`pSmoother`)

The particle smoother function is the novel aspect of SSMs included in package `nLnG`. The pseudocode for the function shown in Algorithm 6 is derived from the written algorithm presented in Section 2.3.2. We will demonstrate the use of the function and its methods with the example tracking problem we have been using throughout this chapter. First let us explain the pseudocode and show how to use the function from the user's point of view.

Looking at Algorithm 6, it begins by showing the user call for the function is `pSmoother()`, the requirements for the function and the inputs the function requires. The user call for the function requires the following inputs: a `pFilterd` object, the parameters, `parms`, the number of realizations to run, `Nreal`, the number of particles to use, `Np`, and any additional information covered by $\cdots$. Note there are also measurement calculations which can be turned on and off by logical statements and we will illustrate them in the example case below. The `pSmoother` function requires that the `pFilterd-nLnG` object contains the filtered states from the `pFilter` results (saved by indicating `save.states=TRUE` in Listing 3.10) and they are just passed through the object to the particle smoother. The `pSmoother` function also ensures

that the number of realizations is greater than zero and that the number of particles is greater than zero (and matches the number of particles for the particle filter procedure performed).

The pseudocode in Algorithm 6 begins with the filter state set, $\{\mathbf{x}^{(Np)}_{1:N|1:N}\}$, being extracted from the `pFilterd-nLnG` object. The filter states are passed inside an outer loop that has been added for the smoother to run the number of realizations indicated by `Nreal`. Once inside the loop, the starting condition (state) for that realization is selected uniformly from the saved filter state set at time N of the particle filter procedure and is stored in $\{\tilde{\mathbf{x}}_N\}$. Then stepping backwards through time starting at $t = N - 1$, the filter state set at time t, $\{\mathbf{x}^{(Np)}_{t|t}\}$, is extracted and used to predict the state at time $t+1$, $\{\hat{\mathbf{x}}^{(Np)}_{t+1}\}$. The difference, $\{\mathbf{dx}\}$, is then taken between the predicted states, $\{\hat{\mathbf{x}}^{(Np)}_{t+1}\}$, and the smoother state at time $t+1$, $\{\mathbf{xs}_{t+1}\}$. The weights, $\mathbf{w}_{t|t+1}$, are then calculated by the `smoother.den` function supplied by the user, computing the density of the differences, $\{\mathbf{dx}\}$, centred at zero. The weights are standardized, then a probability α is generated using a uniform distribution. A sample, $\tilde{\mathbf{x}}_t$, is selected for the realization when its weight, $\mathbf{w}^{(i)}_{t|t+1}$, is greater than the probability α and this continues randomly until one state is selected. The smoother continues doing this each step backwards in time until $t = 1$. Then each realization, $\tilde{\mathbf{x}}_{1:N}$, from the smoother is recorded in the list, $\tilde{\mathbf{x}}_{real}[[nr]]$. Statistics can then be performed on the realizations to come up with means, variances, and, in the particle smoother case, also the quantiles of the smoother distributions. Now that we have introduced the algorithm for the particle smoother in the package `nLnG`, let us show how to use the function.

The Particle smoother function requires that the user supplies a function for the `smoother.den` function attribute in the `nLnG` object. This function calculates the densities of the differences in the `pSmoother` function. The good thing is that the function that calculates the weights for the `pFilter` can also be used for calculating the weights for the `pSmoother`. The `track.obsev.den` function can be used again

Algorithm 6 `pSmoother(object,parms,Nreal,Np,...)`

Require: pFilterd object, filtered states, parameters, Np, Nreal

Ensure: $Np > 0$, $Nreal > 0$

$\{\mathbf{x}_{N|N}^{(Np)}\} \leftarrow$ `save.state(pFilterd object)`

for nr in 1 **to** $Nreal$ **do**

$ip \leftarrow$ `runif(1,Np)`

$\{\tilde{\mathbf{x}}_N\} \leftarrow \{\mathbf{x}_{N|N}^{(ip)}\}$

for $t = N - 1$ **to** 1 **do**

$\{\mathbf{xp}^{(Np)}\} \leftarrow \{\mathbf{x}_{t|t}^{(Np)}\}$

$\{\hat{\mathbf{x}}_{t+1}^{(Np)}\} \leftarrow p(\tilde{\mathbf{x}}_{t+1}|\mathbf{x}_t^{(i)} = \{\mathbf{xp}^{(Np)}\}, \theta)$ (`state.sim`)

$\{\mathbf{xs}_{t+1}^{(Np)}\} \leftarrow \{\tilde{\mathbf{x}}_{t+1}\}$

$\{\mathbf{dx}\} \leftarrow \{\mathbf{xs}_{t+1}\} - \{\hat{\mathbf{x}}_{t+1}\}$

$\mathbf{w}_{t|t+1} \propto \mathbf{w}_t^{(i)} p(\tilde{\mathbf{x}}_{t+1}|\mathbf{x}_t^{(i)} = \{\mathbf{dx}\}, \theta)$ (`smoother.den`)

$\mathbf{w}_{t|t+1} \leftarrow \mathbf{w}_{t|t+1}^{(i)} / \sum_{i=1}^{j} \mathbf{w}_{t|t+1}^{(i)}$

$\alpha \leftarrow$ `runif(0,1)`

$rd \leftarrow$ `runif(1,Np)`

if $\mathbf{w}_{t|t+1}^{(rd)} > \alpha$ **then**

$\tilde{\mathbf{x}}_t = \hat{\mathbf{x}}_t^{(rd)}$

end if

end for

$\tilde{\mathbf{x}}_{real}[[nr]] \leftarrow \tilde{\mathbf{x}}_{1:N}$

end for

(Listing 3.9) for the `smoother.den` function, therefore the user only has to define the density function once. Including the density function to the `nLnG` object is simply done by recalling `nLnG` object with the `smoother.den` attribute defined. Adding the function to the example `nLnG` object can be seen in Listing A.12 and it shows that we have added the `track.obsev.den` function to the `pFilterd-nLnG` object `track.pf`

(the function also could be just specified in the original object if you plan to do particle smoothing). Now that we have added the `smoother.den` calculator we can run the `pSmoother` with the example tracking problem.

The `pSmoother` function requires that the object being specify is a `pFilterd-nLnG` object and that the `pFilterd-nLnG` object has `save.states=TRUE` to record the filter densities, $p(\mathbf{x}_t|\mathbf{Y}_t)$ (the function will give an error if it has not been). The call to the `pSmoother` function for the example tracking data can be seen in Listing 3.13 and it shows that we have supplied the `nLnG` object, `track.pf`, from the particle filter procedure in Section 3.5. Also the user supplies the parameter vector, `theta`, the number of realizations, `Nreal=100`, and the number of particles, `Np=1000` (which must match the number which was run in the particle filter). The rest of the input values for the `pSmoother` are logical attributes for turning on calculations and for recording quantities throughout the process. The `smoother.mean` and `smoother.var` attributes turn on the calculation of the mean and variance from the realizations of the smoother densities. The `save.real` attribute tells the smoother to record the realizations, which is important for quantile calculations in the diagnostic function and the `xhat` attribute records the predicted values from each of the realizations of the smoother. Running the `pSmoother` function will create the object, `track.ps`, in Listing 3.13 and the object will be a new type, `pSmootherd-nLnG`. Now that we have illustrated how to use the `pSmoother` function we should look at how to perform the diagnostics for the results from the test.

```
track.ps <- pSmoother(track.pf,parms=theta,Nreal=100,Np=1000,
                          smoother.mean=TRUE,smoother.var=TRUE,
                          xhat=TRUE,save.real=TRUE
             )
```

Listing 3.13: Calling the `pSmoother` function on example tracking data, running 100 realizations, with 1000 particles and recording all the quantities from the procedure.

The user can access the different result quantities stored in the `pSmootherd-nLnG` object using the class methods. The smoother mean and variance can be extracted by using the two class methods `smooth.mean` and `smooth.var` respectfully for the `pSmootherd` object. For the example tracking problem, the smoother mean can be extracted and plotted against the true state positions and the observations to produce the 2D plot in Figure 3.6. Figure 3.6 shows that the smoother mean (green) again follows much closer to the true state (blue) positions than to the observations (red) positions. One thing to notice is that the smoother mean does not react as strongly to the observations as we saw with the filter; this feature is key in the smoother producing a much less rough looking estimation of the true state than the filter produces. For quick visualization of the procedure results, the user can also use the `smoother.diag` diagnostic function with the `pSmootherd` object. The call to the `smoother.diag` function for the example tracking data can be seen in Listing A.13 in Appendix A. From the results above and the findings in the Appendix, again we can say the particle smoother did good job of recovering the true state of the system. With comparison to the recovered state estimate from the particle filter, the state estimate has a much more smoother approximation of the of the true state (See Appendix B).

We have detailed the algorithm for the `pSmoother` function in the `nLnG` package and illustrated the function's use with the example tracking problem. We can now move on to the parameter estimation methods available in `nLnG`. We will start first

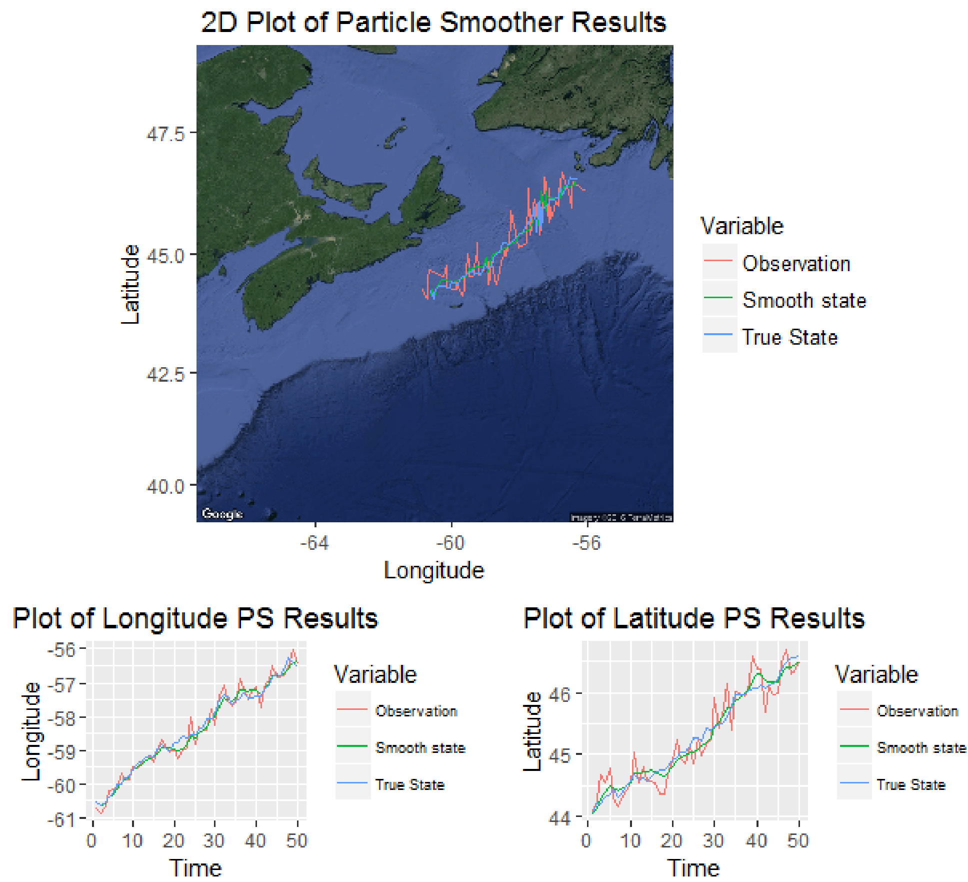


Figure 3.6: In the top plot in the figure, the smoother states (red), observed states (green) and the true state (black) are plotted against each other to form the 2D plot from the Particle smoother results on the Scotian Shelf map. Underneath the 2D plot each of the longitude and latitude variables are plotting individually with the smoother states (red), observed states (green) and the true state (black) being plotted against the time

with maximum likelihood estimation (MLE).

3.8 MLE Methods

In this section we will illustrate the use of both maximum likelihood estimation (MLE) methods that are included in the package nLnG. The MLE methods aid in the process of parameter estimation for the model used to represent real world phenomenon. We will demonstrate the MLE methods with the example tracking problem that has been used throughout this chapter. But first let us talk about how the MLE methods are used in the design for nLnG, then explain the algorithms for each MLE method and show the user how to obtain the value(s) using the built in package methods, or access the raw data from the object itself. As noted in Section 2.4.1, the MLE procedures included in nLnG do not return optimized MLE values, but instead the estimate, in terms of the log-likelihood value, for the data using the parameter(s).

The MLE methods included in the nLnG package are not procedures (or functions) by themselves, but are included as the main result from the function for the two filtering methods available in nLnG (both demonstrated earlier in this chapter). The two filtering methods each return only the MLE unless additional measurement calculations are specified by the user. The MLE is returned in terms of the log-likelihood value for the data in terms of the parameter(s) used for the procedure. The methods for calculation of the log-likelihood for each filter are slightly different from each other. The particle filter sums the weights from each of the measurement steps, whereas the Kalman filter calculates the innovation error for each time-step, (both methods were illustrated in Section 2.14). The algorithms for each method are included in each of the filter Algorithms 3 and 4, presented earlier in Chapter 3 (illustrated in the algorithms by the calculation of $l_t(\theta)$ during each step of the filter and the summation of the log-likelihood, $l(\theta)$ at the end of filtering in each case). As mentioned in Chapter 3, the MLE methods included in the filtering procedures in package nLnG could also have been included in the smoothing procedures as well.

Though the log-likelihood estimates obtained from the smoothing procedure would just be redundant information as it is already calculated by running the filter (which is required to be performed before any smoothing is done). With most of the details for the MLE methods presented earlier, we can move on to explaining how the user can access the MLE methods in `nLnG`.

The log-likelihood value can be obtained from either the filtered object by using the method (or helper) function `logLik` on the object the filtered results are saved in (shown in Listing 3.8 and 3.11), or the user can also pull out all the conditional log-likelihood values from each of the time-steps so that they can be inspected by using `@loglik` on the filtered object. The `logLik` function just sums each of the conditional log-likelihood time- steps value into the full log-likelihood for the procedure. Now that we have detailed how the package `nLnG` performs MLE and how log-likelihood values are obtained, we can move on to demonstrating MLE with the example tracking problem.

The general practice of parameter selection for either of the state or observation model is usually more involved than this demonstration will be and we will see this in Chapter 4 when working with the real data set. MLE methods help in the process of parameter estimation by maximizing the log-likelihood estimate when the parameter values being used best fit the modelled phenomenon. For the illustration of how the log-likelihood is maximized, we will be running both the filtering methods to profile the likelihood rather than maximizing it and computing log-likelihood estimates for different values of the dynamical parameter, `d` (or `A` in the Kalman filter case) for the example tracking problem, while holding all other parameters constant.

The procedure for performing this can be seen in Listing A.17, in Appendix A. In Listing A.17, for the test the dynamics parameter vector, `dvec` is set to be the sequence from 0.997 to 1.003 by 0.001 increments. Variable vectors (`loglikest` and `loglikestKF`) are set up and used to collect the log-likelihood from each run of the particle filter and Kalman filter, as the filters are looped over the sequential values

of the dynamics parameter. The results from the procedure are summarized in the following Table 3.5.

Dynamics Parameter Value	**Kalman Filter Maximized log-likelihood**	**Particle Filter Maximized log-likelihood**
0.997	-91.24812	-182.72203
0.998	-53.24622	-67.31155
0.999	-31.62655	-30.23498
1.000	-26.38947	-22.24077
1.001	-37.53413	-39.2682
1.002	-65.05835	-98.78089
1.003	-108.95874	-255.49694

Table 3.5: Values for the profiled MLE (log-likelihood) over the range of dynamics parameter listed for the example tracking problem using both the Kalman filter and particle filter procedures.

The table shows that at the two ends of the dynamic parameter value range, the Kalman filter and particle filter log-likelihood estimates are different. This occurs from the particle filter's effective sample size growing smaller as the parameter value is getting farther away from the truth, resulting in a effective sample size that is made up of fewer unique particles. This is a results of the particle filter likelihood being affected by Monte Carlo variation. As the dynamics parameter approaches the true value the log-likelihood values from the two filters are much closer to each other. It shows that both filters maximize the log-likelihood for the true value of the dynamics parameter at 1.000 and this is what we are expecting to see with the example tracking problem (knowing the true parameters in advance). Now randomly searching or profiling the likelihood is not an effective method for parameter estimation when trying to parametrize the SSM, but the general idea of maximizing the log-likelihood will be used in the parameter estimation methods yet to be demonstrated in package nLnG and this was a quick way in which to illustrate this concept.

We now have illustrated the MLE methods available for package `nLnG` and demonstrated their use in the parameter estimation case, we will move on to the next parameter estimation method included in `nLnG`, know as state augmentation. State augmentation is the first method which allows for time varying in the parameter space for the SSM and utilizes the particle filter as the computational engine.

3.9 State Augmentation (`sAugmentation`)

The state augmentation parameter estimation method is a special inclusion in the `nLnG` package and is mostly made available to highlight the issues with this type of estimation, as well as motivate the next parameter estimation method, the multiple iterated filter. In this section we will introduce the pseudocode for the `stateAugmentation` function included in package `nLnG` and demonstrate the function's use for the reader using the example tracking problem to illustrate some issues with this type of estimation. Let us first introduce the pseudocode for the algorithm.

The `stateAugmentation` function is more generally a wrapper function for the `pFilter` function that allows the user easier set-up for the augmentation settings. The algorithm looks very similar to the `pFilter` function in Algorithm 4, since state augmentation just uses the particle filter procedure, but allowing for the parameter(s) to be able to time vary. With only slight changes, the algorithm for state augmentation will not differ greatly from Algorithm 4 for the particle filter and we will highlight just the modifications. The pseudocode for the `stateAugmentation` function can be seen in Algorithm 7.

In Algorithm 7, it first shows the user call for the `stateAugmentation` function. It looks very similar to the user call for the `pFilter` function we have seen, besides the following additions to the call. The extra requirements are the parameter(s) which vary with time, contained in a named vector, `pars`, and the standard deviation for

each parameter(s), contained in named vector, `pars.sd`. The user should also set the maximum failures, `max.fail`, to allow for more failures of the filtering during estimation (by default `max.fail` will be set to 100 with a warning, if not specified). The `stateAugmentation` function also requires all the same base inputs as the `pFilter` function, specifying the `nLnG` object, the parameters, `parms` (initial parameters in this case), and the number of particles, `Np`. The `stateAugmentation` function also has logical variables for the collection of the predictive mean and variance, filter mean and variance. The parameter values for the augmentation are added to both sets of the means and variances. Let us move on to look at the main body of the algorithm for the `stateAugmentation` function in Algorithm 7.

The main body of the algorithm resembles the particle filter version in Algorithm 4 for the most part and for the sake of not being redundant we will highlight the differences made to the state augmentation algorithm. The first thing the user should notice is that the `stateAugmentation` algorithm requires the user to supply the parameter(s) to vary, `pars`, and the standard deviation for the varying parameter(s), `pars.sd`. If either are not supplied, an error will prompt the user that no parameter(s) were specified. Notice that the parameters supplied to the function are the initial state of the parameters, θ_0, some of which (the parameters defined in `pars`) will evolve with each time-step. The initial state, $\dot{\mathbf{x}}_t$, has now been edited to include the parameter, θ_t, making it equivalent to the Equation 2.23, and the error, $\dot{\mathbf{e}}_t$, includes the error for the parameter(s), $\mathbf{e}_t^\theta$, to make it equivalent to Equation 2.26. The parameter space is represented by a set of particles equal to the number, `Np`, supplied by the user, the same way the state is represented. The method for calculating/updating of the predictive and filter states is the same as was seen in the particle filter
algorithm, with the only difference being that the ensemble being used in state augmentation includes the parameter(s). The prediction step moves forward the state(s) and parameter(s) in time from $t-1$ to t and then the measurement step computes the weights (or likelihoods) for the state(s) and parameter(s). The parameter(s) are

Algorithm 7 `stateAugementation(object,parms,Np,pars,pars.sd,mif.fail,...)`

Require: `nLnG object`, `Np`, `parms`, `pars`, `pars.sd`

Ensure: `Np` > 0, `pars.sd` > 0

$\theta_0 \leftarrow \theta_{init.cond}$, $\mathbf{x}_0 \leftarrow \mathbf{x}_{init}$,

$\dot{\mathbf{x}}_t^{'} = \begin{bmatrix} \mathbf{x}_t & \theta_t \end{bmatrix}$, $\dot{\mathbf{e}}_t^{'} = \begin{bmatrix} \mathbf{v}_t & \mathbf{e}_t^{\theta} \end{bmatrix}$

for t in $1 : (N-1)$ **do**

$\{\dot{\mathbf{x}}_{t|t-1}^{(j)}\} \leftarrow p(\dot{\mathbf{x}}_t \mid \dot{\mathbf{x}}_{t-1} = \{\dot{\mathbf{x}}_{t-1|t-1}^{(j)}\}, \theta_{t-1|t-1})$ (`state.sim`)

$w_t^{(j)} \leftarrow p(\mathbf{y}_t|\dot{\mathbf{x}}_t = \dot{\mathbf{x}}_{t|t-1}^{(j)}, \theta_{t-1|t-1})$ (`obsev.den`)

$w_t^{(j)} \leftarrow w_t^{(j)} / \sum w_t^{(j)}$ (`pFilter_computations`)

$\theta_{t|t-1}^{(j)} \leftarrow \theta_{t-1|t-1}^{(j)} + N(0, \texttt{pars.sd})$ (`pFilter_computations`)

$l_t(\theta_t) \leftarrow log(Np^{-1} \sum_{j=1}^{Np} w_t^{(j)}) = log\ p(\mathbf{y}_t|\mathbf{y}_{t-1}, \theta_t)$ (`pFilter_computations`)

$\hat{\dot{\mathbf{x}}}_{t|t-1} = \frac{\sum_j \dot{\mathbf{x}}_{t|t-1}^{(j)}}{J}$ (`pFilter_computations`)

$\hat{\sigma}_{\mathbf{x}_{t|t-1}} = \frac{\sum_j \dot{\mathbf{x}}_{t|t}^{(j)} \dot{\mathbf{x}}_{t|t}^{(j)} - (\sum_j \dot{\mathbf{x}}_{t|t-1}^{(j)})/j}{J-1}$ (`pFilter_computations`)

$\hat{\dot{\mathbf{x}}}_{t|t} = \frac{\sum_j \dot{\mathbf{x}}_{t|t-1}^{(j)} w_t^{(j)}}{\sum_j w_t^{(j)}}$ (`pFilter_computations`)

$\hat{\sigma}_{\dot{\mathbf{x}}_{t|t}} = \frac{\sum_j \dot{\mathbf{x}}_{t|t}^{(j)} w_t^{(j)} \dot{\mathbf{x}}_{t|t}^{(j)} - (\sum_j w_t^{(j)} * w_t^{(j)}) / \sum_j w_t^{(j)}}{\sum_j w_t^{(j)}}$ (`pFilter_computations`)

for j in $1 : J$ **do**

while $U_j(0, (w_t^{(j)}/Np)) > c_p$, where $c_p = \sum_m w_m$ **do**

set $p = p + 1$, where $p_1 = 1$

end while

set $I_j = p$, where I is the re-sampled index

end for (`pFilter_computations`)

re-sample $\{\dot{\mathbf{x}}_{t|t-1}^{(j)}, w_t^{(j)}\}$ according to I_j

$\{\dot{\mathbf{x}}_{t|t}^{(j)}\} \sim p(\dot{\mathbf{x}}_t|\mathbf{Y}_t, \theta_{t|t})$

end for

$l(\theta_t) \leftarrow \sum_{t=1}^{N} l_t(\theta_{t|t})$

predicted forward with the noise being described by a Normal distribution centered at zero and varied by the supplied parameter standard deviation(s), `pars.sd`. The state(s) and parameter(s) are then re-sampled with replacement according to those weights, to create the new ensemble of state(s) and parameter(s). The log-likelihood, $l(\theta_t)$ is then summed for the the procedure and returned. This details the main differences made for the `stateAugmentation` algorithm to allow for time varying in the parameter space. Let us now demonstrate the function's use in parameter estimation for SSMs.

For this demonstration we will be using the example animal tracking problem that has being used throughout Chapter 3 for introducing the functions in `nLnG`. Again the test will be estimating the value of the dynamic parameter, $\mathbf{d}_t$, the same as was done for the MLE parameter estimation examples, only this time by letting the parameter(s) varying with time during the process. In Listing 3.14, the user call for the demonstration can be seen for the `stateAugmentation` function. The user call is very similar to the `pFilter` function, with the following exceptions. The parameter for estimation is specified by setting the parameter in a named vector, `pars=c("d")`. A standard deviation for the parameter is given by `pars.sd=c(d=0.005)` and a maximum filter failures is specified, `max.fail=100`. The other defined attributes are the same as was seen with the `pFilter` user call, using the `nLnG` object, `track.ex`, the original parameters, `theta`, the number of particles, `Np`, and the logical attributes for predictive and filter means and variances (which now contain the parameters as well). Let us now move on to look at the results from the test.

```
track.sa <- stateAugmentation(track.ex,parms=theta,Np=1000,
                              pars=c("d"),pars.sd=c(d=0.005),
                              pred.mean=TRUE,pred.var=TRUE,
                              filter.mean=TRUE,filter.var=TRUE,
                              save.states=TRUE,save.parms=TRUE,
                              max.fail=100)
```

Listing 3.14: User call for the `stateAugmentation` function with the Tracking example data, estimating the dynamics parameter,`d`, with a standard deviation of 0.005 and saving the finding for the predictive and filter means and variances.

The parameters sample space can be extracted from the object with the saved result (`track.sa` here) to produce a credible region for the estimated parameter and is done by pulling `save.parms` from object using the `@`. The time-varying parameter can be pulled from the lists and plotted to give the plot below, Figure 3.7. In Figure 3.7 the estimated dynamics parameter starts at 1.00 and ends at around 1.0008667. The issue with this is that while the median is very close to the initial and true value of the parameter, `d`, the trace of the prediction contains deviations from the true value and there is no indication of what the estimated value should be. The returned log-likelihood value for the test is -42.93537 (Listing A.18), which is quite different from the value seen in Listing 3.11, -22.41174, for the true value. Figure 3.7, shows the 95% credible region for the prediction median as well, the width of the region is approximation 0.02 wide for the majority of the time-steps. This again leaves a large region of values in the credible region for the estimate of the parameter and is not helpful in the determination of the estimated value to use from the test. Confounding the problem is if the test is run again the values could easily be different. This is the biggest problem with using state augmentation for parameter estimation for SSMs and why `nLnG` will include the MIF parameter estimation procedure that

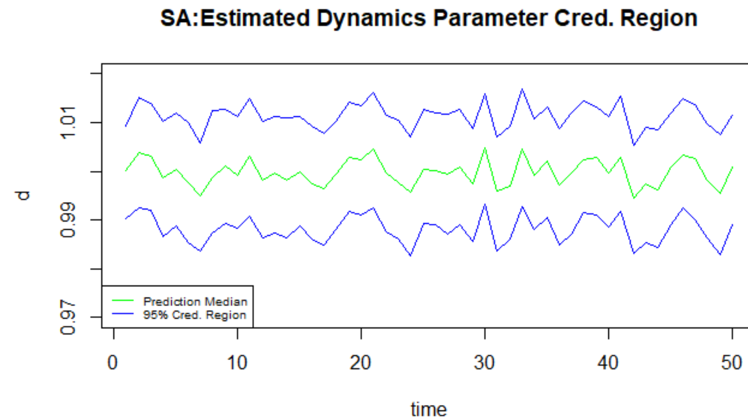


Figure 3.7: The plotted sample estimate of the dynamics parameter median from the `stateAugmentation` function test, with 95% credible intervals.

corrects for this flaw.

This finishes the illustration using the `stateAugmentation` function with the example tracking problem. The demonstration highlighted issues with using this method of parameter estimation for parametrizing models and motivated the need for the next parameter estimation method to be covered in package `nLnG`. We next move on to the parameter estimation method known as MIF.

3.10 Multiple Iterative Filtering (`mif`)

The parameter estimation method known as multiple iterative filtering (MIF) is the last of the main procedures to be included in package `nLnG`. It builds on both the previous demonstrated parameter estimation methods (MLE and state augmentation) and offers the user a method for parameter estimation. In this section we will be introducing the reader to the use of the `mif` procedure in `nLnG`, detailing the pseudo code for the function's algorithm and demonstrating the procedures with the example

tracking problem. We will first detail the pseudo code for the algorithm and define the user call for the `mif` function in `nLnG`.

The pseudocode for the `mif` procedure is adapted from Ionides et al. (2006) written algorithm in Section 3.10 for package `nLnG` and can be seen in Algorithm 8. The algorithm first shows the function call for the `mif` function. In the function call, the user must supply an `nLnG` object to the procedure. The `nLnG` object requires only that the state and observation models attributes be specified in the object, the same as we seen with the `pFilter` function. The `mif` procedure makes use of the `pFilter` procedure as the engine for its calculation. Next the user needs to supply the starting positions for all the parameters being used in the procedure in `start` (usually a initial guess for the parameter(s) being estimated). Now depending on which type of estimation operation the user wishes to run, the parameters are specified in one of two ways. If you want to perform ordinary parameter estimation with the `mif` operation, the parameters are supplied to the function through the `pars` attribute in a character vector naming the parameters. The other option is if the user wishes to perform fixed-lag smoothing for the estimation of initial-value parameter(s) (IVP(s)). The user supplies the IVPs through the `ivps` attribute by a character vector containing only the parameter variable name(s) to be estimated. The user must supply the standard deviation for the RW through the `rw.sd` attribute, which is applied to the parameter(s) for either of the two cases of estimation by a named vector with the size of the deviation to use. In the case where the estimating parameters are supplied by `pars` and `rw.sd`, they must match, and be defined by a positive standard deviation, and are used in the RWM for the parameters in `pars`. In the case where we are estimating IVP(s) the parameters in both `ivps` and `rw.sd` must match, having a positive error value and are used as a scale factor for scaling the IVPs. The parameters defined in either of `pars` or `ivps` should be a subset of the parameters supplied by `start` (we will demonstrate this shortly with the example problem). The user must specify the number of MIF iterations to perform, `Nmif`, and

the number of particles, Np, for the pFilter to use in the ensembles. Now that we have explained how to specify the parameters for using the mif function and set the iterations, we still need tell the function how to decrease the parameter variance.

For controlling the variance for the procedure, the user first selects the scaling coefficient for the width of the starting particle distributions by supplying a positive integer for the variance factor, var.factor, for the parameters contained in the rw.sd vector. The user then must specify the exponential cooling factor, α, by setting cooling.factor attribute to be a positive number no greater than one. The cooling factor determines how fast we decrease the variance between each MIF iteration. For the case where we are estimating IVPs, the user is required to supply the time-point to apply the fixed-lag smoothing by specifying the ic.lag attribute. The IVPs are updated/replaced with their filtering means at this time. The ic.lag value needs to be a positive integer and should be less than the last time-point in the series (or the mif function will automatically set it to be the last time-point for the series with a warning to the user). The user may also supply their own function to set up the starting particle distribution by creating a function of type particles and assigning it to the same attribute name in the function call. If particles is not specified by the user, the default particles function samples from a multivariate normal distribution with mean, center, and standard deviation, sd is used. The other optional user input is the method used for the update rule in the procedure and can be set to one of the three options by the method attribute. The user can pick between using Ionides et al. (2006) iterating filtering rule by setting method="mif", or updating the parameter to the unweighted average of the filtering means of the parameters at each time by setting method="unweighted", or updating the parameters to the filtering means at the end of the time series by setting method="fp". By default the mif function uses the method="mif" update rule. Now that we have finished detailing the user inputs for the mif procedure in nLnG, we will now describe how the mif algorithm itself operates.

Looking now at the procedure for the `mif` function in Algorithm 8, the first thing we see is the required arguments that the user must supply to run the MIF. Like the rest of the procedures in package `nLnG` the `mif` function requires an `nLnG` object containing the necessary state and observation equations to run the `pFilter` function (See Section 3.5). The number of MIF iterations to make, `Nmif`, must be supplied. Next the parameters, θ, are required to be supplied through the methods described earlier depending on the type of estimation being performed. The cooling or discount factor, α, and the variance multiplier, c, are both required for the algorithm and lastly the number of iterations for the particle filter, Np, is required. The Algorithm next shows hows the input values for the `mif` function link to the symbols used to describe the procedure. The start values for the parameters in `start` are saved in θ, the variance cooling, factor `cooling.factor`, are saved in α, and the variance multiplier, `var.fact`, are saved in c. After the variable setup we get the main part of the MIF Algorithm.

The main part of the Algorithm starts with entering the primary loop, looping from n to the number of MIF iterations, $Nmif$, for the procedure. For each iteration of the loop the MIF algorithm starts by computing the cooling factor, α, raised to the power of $n-1$ (the number of the current iteration minus 1) and storing it as α and also as γ, by squaring the α value. Next in the Algorithm the cooled sigma, σ_n, is updated by multiplying α by the standard deviations for the parameters being estimated, σ. Now that we have the cooled sigma, the parameters are next initialized with it and the variance factor, c. The new parameter set is drawn for $\hat{\theta}^{(n)}$, using θ as the center and σ_n multiplied by c as the standard deviation for the parameter space and $\hat{\theta}^{(n)}$ is then used by the `pFilter` function. The `pFilter` function call in the `mif` algorithm shows the particle filter runs a state augmented version of the filter. The cooled sigma, σ_n, is used by the `pFilter` function to perform the random walk for the parameters (`.rw.sd=sigma.n`) and $\hat{\theta}^{(n)}$ is set as the `parms` vector. The logical arguments for the prediction variance and the filter mean are set to `TRUE`, they

are key results from the `pFilter` function for the update rule in the `mif` algorithm. The running of the `pFilter` function computes the results of Equations 2.28 and 2.29 from the MIF Section 2.4.3. Depending on what type of MIF update rule was selected by the user one of three different update rules, use the results from the `pFilter` to update the variance, V, and $\hat{\theta}_t^{(n)}$.

Algorithm 8 `mif(object,Nmif,Np,start,pars,ivps,cooling.factor,var.fact,...)`

Require: nLnG object, Nmif, Np, parameters, θ, α, T, c

$\hat{\theta} \leftarrow$ `start`, $\alpha \leftarrow$ `cooling.factor`, $c \leftarrow$ `var.fact`, $\sigma \leftarrow$ `.rw.sd`

for n in 1 to $Nmif$ **do**

$\alpha = \alpha^{n-1}$, $\gamma = \alpha^2$ (`mif.cooling(`α`,n+1)`)

$\sigma_n = \alpha * \sigma$

$\hat{\theta}^{(n)} = \hat{\theta}_t(\hat{\theta}^{(n)}, \sigma_n * c)$ (`particles(`$\hat{\theta}^{(n)}$`,`σ_n`*c)`)

$\{\mathbf{x}_{t|t}^{(j)}, \hat{\theta}_t^{(n)}\} \leftarrow$ `pFilter(object,Np,`$\hat{\theta}^{(n)}$`,`$\cdots$`)`

if method = "mif" **then**

$V_{t,n} = V_t(\hat{\theta}^{(n)}, \sigma_n)$

$V_{1,n} = \gamma * (1 + c^2) * \sigma^2$

$\hat{\theta}_t^{(n)} = \hat{\theta}_t(\hat{\theta}^{(n)}, \sigma_n)$

$\hat{\theta}^{(n+1)} = \hat{\theta}^{(n)} + V_{1,n} \sum_{t=1}^{T} V_{t,n}^{-1}(\hat{\theta}_t^{(n)} - \hat{\theta}_{t-1}^{(n)})$

else if method = "unweighted" **then**

$\hat{\theta}_t^{(n)} = \hat{\theta}_t(\hat{\theta}^{(n)}, \sigma_n)$

$\hat{\theta}^{(n+1)} \leftarrow \frac{\sum_{t=1}^{T}\{\hat{\theta}_t^{(n)}\}}{T}$

else if method = "fp" **then**

$\hat{\theta}_t^{(n)} = \hat{\theta}_t(\hat{\theta}_T^{(n)}, \sigma_n)$

$\hat{\theta}^{(n+1)} \leftarrow \hat{\theta}_t^{(n)}$

end if

end for

$\hat{\theta}^{(N+1)} = \hat{\theta}^{(n+1)}$

If the user selects the `mif` method of update, the MIF rule is applied from Ionides et al. (2006). This method first sets $V_{t,n}$ equal to the prediction variance from the `pFilter` function for the parameters. It then calculates $V_{1,n}$ by multiplying γ by 1 plus c squared and then multiplying that value by the variance of the parameters being estimated, σ^2. The filter density for the parameters from the `pFilter` function result is extracted and then saved as $\hat{\theta}_t^{(n)}$. The parameters, $\hat{\theta}^{n+1}$, are then updated by adding the previous parameters $\hat{\theta}^n$ to $V_{1,n}$, multiplied by the sum of the differences of $\hat{\theta}^n$ divided by the transpose of the $V_{t,n}$. If the update rule was selected as `unweighted` by the user, then $\hat{\theta}^n$ is set to the filter mean from the `pFilter` function for the parameters. $\hat{\theta}^{n+1}$ is then updated by taking the means of $\hat{\theta}^n$ for each of the parameters. In the last update rule, `fp`, the user can select to set $\hat{\theta}^n$ to the filter mean from the `pFilter` function for the parameters at the final time-step. The parameters, $\hat{\theta}^{n+1}$, are then updated by just being set equal to $\hat{\theta}^n$. This details the end of the different update rules for the `mif` function. When the last iteration is reached for the parameters, $\hat{\theta}^{n+1}$, is taken to be the MLE of the parameters for the fixed parameter model $\hat{\theta}^{N+1}$ (Ionides et al., 2006). Now that we have finished detailing the algorithm for the `mif` function in `nLnG`, we can move on to illustrating its use in parameter estimation for SSMs.

For this demonstration of the `mif` function, we will again be using the example tracking problem and be estimating the dynamics parameter, $\mathbf{d}_t$. We will be discussing both the ordinary and IVP parameter estimation methods for the `mif` function. We should first look at the user call made for the `mif` function using the example tracking problem and then do an analysis of the results from the test. The set-up for the `mif` function can be seen in Listing A.19. It shows that the parameters from the particle filtered object, `track.pf`, are saved into the variable, `truth`, using the `coef` method in the `nLnG` package. The dynamics parameter, `d`, is saved into the variable, `parEst`, as a character vector. This set-up will be used for both the ordinary and IVP parameter estimation methods. The user call for the `mif` function

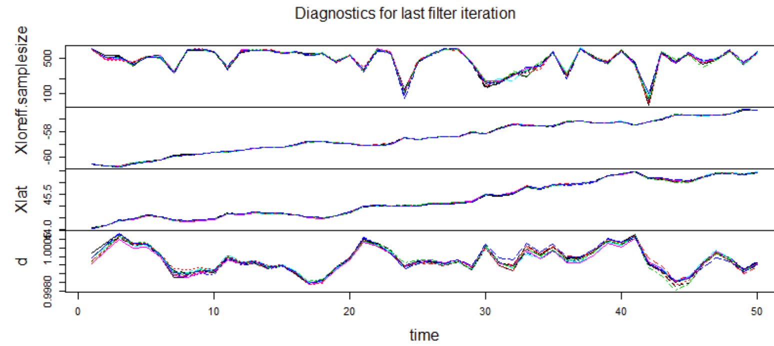
for ordinary parameter estimation can be seen in Listing A.20, and for IVP parameter estimation can be seen in Listing A.21. The two user calls for the `mif` function look very similar, with one difference to indicate the type of estimation to perform (explained above). For this reason we will do one explanation of the user call and highlight the lone difference between them. The first thing to notice in the two Listings is that the `mif` function call is enclosed inside a `replicate` function. By doing this we are running the `mif` function multiple times by starting the dynamics parameter at various different locations. This is just to illustrate the convergence of the MIF method to the same value from different starting values. The test is performed 10 times for the illustration, as `n=10` in the Listings. The original parameters are next saved into the variable `guess` from the variable `truth` and then the dynamics parameter is varied by using a normal distribution centred at the original value of the dynamics parameter, `truth`, with a standard deviation of 0.005. For the `mif` function user call in the Listings, the following attributes are defined. The particle filtered object, `track.pf`, is supplied, since it has all the necessary parts needed to run the particle filter in the MIF procedure. The number of MIF iterations, `Nmif`, is set to 50, and the initial values for the parameters, `start`, is given the parameter vector, `guess`. This is where the two parameter estimation methods differ. In Listing A.20, for ordinary parameter estimation the attribute `pars` is supplied the variable created in Listing A.19, `parEst`, containing the parameter for estimation. In Listing A.21, for IVP parameter estimation the attribute `ivps` is supplied the variable, `parEst`. The standard deviation for the parameter RW is set by the variable, `rw.sd`, and set to `d=0.001`. Each of the two parameter estimation methods pulls the RW standard deviation from this variable. The number of particles for the particle filter to use, `Np`, is set to 1000. The time-point to apply the fixed-lag smoothing, `ic.lag`, is set to the last point of the series (50). The variance cooling factor, `cooling.factor`, is set to 0.99 and the variance multiplier factor, `var.factor`, is set to two. The last attribute in the user call is the maximum failure allowed for the `pFilter` function,

`max.fail`, and it is set to 100, to give the filter a higher threshold on failure for the parameter estimation process. Now that we have described the set up for the tests done for this demonstration, we can move on to examine the results from the `mif` functions in Listing A.20 and A.21.

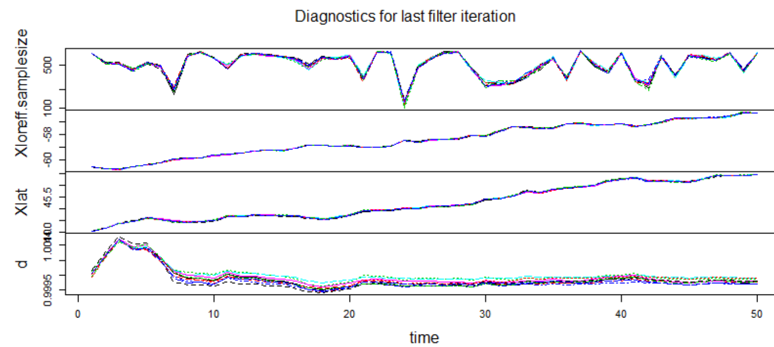
The results from the `mif` function are each saved in variables as list objects, `track.Pmif` in Listing A.20 and `track.Imif` in Listing A.21. The user can pull these objects apart by referencing the variable result lists individually and extract specific parts of the results by using the `@` sign, with the attribute name (ex. in Listing A.23). The tests performed in Listing A.20 and A.21 are large procedures to demonstrate the `mif` function and the objects produced contain the results of all 10 runs of the replicated test. For these situations, the `nLnG` package includes a method called `mif.comp` that allows the user to quickly produce a set of diagnostic plots for the results from the `mif` function and it will be used to do the heavy lifting with the example cases used here. Listing A.22 shows the user call for `mif.comp`. It requires that the `mif` object with the saved results be supplied in the function call (`track.Imif` in this Listing). The method produces a number of diagnostic plots of the keys results from the MIF procedure that would be of interest to the user for analysis of the test. The diagnostic plots produced by the `mif.comp` function for the two examples cases of parameter estimation can be seen in Figures 3.8 and 3.9. The `mif.comp` function produces two sets of diagnostics plots when called. The first set of plots are shown in Figure 3.8 and are the diagnostics for the last filter iterations. The second set of plots are shown in Figure 3.9 and are the MIF convergence diagnostics. In Figure 3.8, the results from the `track.Pmif` object are plotted on the top in Figure 3.8a and the results from the `track.Imif` object are plotted on the bottom in Figure 3.8b. The following plots are produced: a plot of the effective sample sizes, a plot of the the state variable(s) (in this case, one for each of the longitude and latitude states), and a plot for each of the estimated parameter traces (in this example, the dynamics parameter, `d`). In Figure 3.9, the results from the `track.Pmif` object are plotted in 3.9a and

the results from the `track.Imif` object are plotted in 3.9b. Referring to one of the sub-figures, it shows in the convergence diagnostics the following plots are produced: a plot of the log-likelihood convergence, a plot of `nfail` (number of filtering failures) convergence and a plot for each of the estimated parameter(s) convergence. Now that we have introduced the `mif.comp` method, we can now examine the results, for the two `mif` objects being saved in Listing A.20 and A.21.

In Figure 3.8a, the results from the ordinary parameter estimation show that for each of the 10 tests the results for the latitude and longitude states are fairly consistent as each of the traces are plotted pretty much on top of each other. The effective sample size plot shows that the effective particles only drop below 100 twice and is consistent across the 10 tests as well, with little variation. The plot of the dynamics parameter trace shows each of the 10 tests produce close to the same estimation result, with only slight variations in the traces at about the 8, 32 and 44 time-steps. With the results being fairly consistent across the 10 tests, the `mif` function has done a good job of estimating the ordinary parameter, `d`. The convergence diagnostics for the test will be examined shortly to show the reliability of the parameter estimation, but first let us review the IVP test filtering iteration diagnostics. Referring to Figure 3.8b now, the results from the IVP parameter estimation test show again that the `mif` function produces fairly consistent findings for the latitude and longitude states, with each of the 10 traces being on top of each other. The plot of the effective sample size is very consistent across the tests and the particles size only dips low around the 8 and 24 time-step. The biggest difference is in the trace of the estimated parameter, `d`, from the previous results. The IVP parameter estimation focus is on estimating the initial parameter values, therefore after the first iteration the parameter value goes unchanged. The parameter trace moves slightly at the beginning and then levels off within the first five time-steps, this allows for convergence of initial parameter values and is the expected result from the test. Now that we have reviewed the last filter iterations diagnostics, let us move on to the convergence diagnostics produced by



(a) The plotted last filter iteration diagnostics from the ordinary parameter estimation test using the `mif` function for the example tracking data. Plots included in the panels (top to bottom) are the effective sample size, filter state for the longitude, filter state for the latitude, and the estimated parameter, `d`.

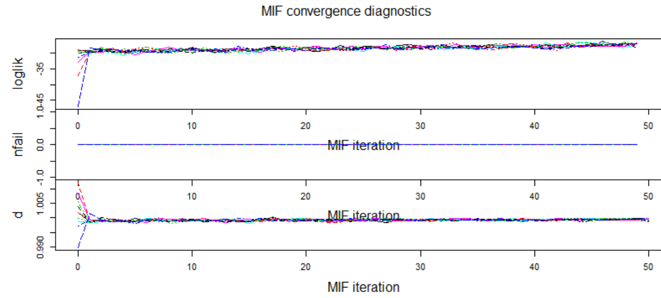


(b) The plotted last filter iteration diagnostics from the IVP(s) parameter estimation test using the `mif` function for the example tracking data. Plots included in the panels (top to bottom) are the effective sample size, filter state for the longitude, filter state for the latitude, and the estimate parameter `d`.

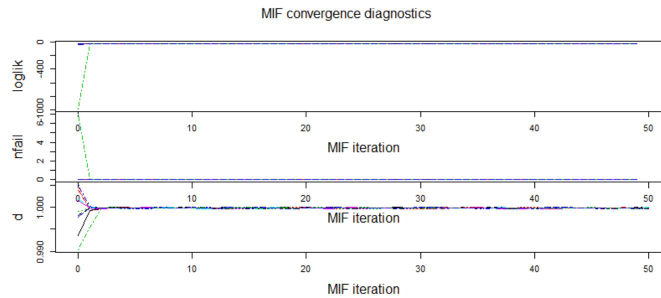
Figure 3.8: The plotted last filter iteration diagnostics from the two `mif` function tests for the example tracking data. a) the ordinary parameter estimation test and b) the IVP parameter estimation test.

`mif.comp`.

In Figure 3.9a, the convergence diagnostics for the ordinary parameter test shows very good results. The plot of the log-likelihood value has converged around the -30



(a) The plotted convergence diagnostics results from the ordinary parameter estimation test using the `mif` function for the example tracking data. Plots included in the panels (top to bottom) are estimated log-likelihood, the number of filter failures and the estimated parameter, `d`.



(b) The plotted convergence diagnostics results from the IVP parameter estimation test using the `mif` function for the example tracking data. Plots included in the panels (top to bottom) are estimated log-likelihood, the number of filter failures and the estimated parameter, `d`.

Figure 3.9: The plotted onvergence diagnostics results from the two `mif` function tests for the example tracking data. a) the ordinary parameter estimation test and b) the IVP parameter estimation test.

mark, even though it starts at numerous places on the y-axis. This value is close to the value we saw with the particle filter test in Section 3.5 and the difference between them can be attributed to the parameter changing with time in the MIF test. Next the plot of `nfail` shows that each of the 10 tests had no filtering failures. This can be expected since we knew the true parameter to begin with and the size of the error

used was small. The parameter convergence is even close to the true value of 1.00, with all 10 tests converging around that mark or just slightly below it. From the findings, the `mif` function has done a very good job of recovering the true value of the estimated parameter, `d`. Referring now to Figure 3.9b, the convergence diagnostics for the IVP test are quite different then what we just saw for the ordinary estimation. The plot of the log-likelihood convergence shows a value below zero, though with the green line within the plots has taken longer to converge. Besides the green line that only had failures in the first time-step, the plot of `nfail` convergence looks like most of the 10 tests had zero failures. The convergence plot for the parameter, `d`, shows that all but the green line converge in the first time-step to the value of 1.00 for the rest of the MIF iterations. The IVP parameter test has done a good job of recovering of the initial parameter value of 1.00 used to simulate the data. Now that we have demonstrated how to quickly access the results of the `mif` function and analysed the performed tests, let us move on to seeing how the ordinary parameter estimation did numerically.

In Table 3.6, the numeric results for the ordinary `mif` parameter test are listed. The reader can refer to Listing A.24 to view the procedure for obtaining the results to Table 3.6. The Table lists the estimated parameter, `d`, the log-likelihood, and the log-likelihood standard error for both the true values of the parameter and the estimate from the `mif` test. Looking at the Table, it can be seen that the estimate for the parameter value is fairly reasonable in comparison to the true value. The estimate of `d` is only off the true value by 0.0006596. The log-likelihood values differ by -2.2429 from each other and the standard errors are as they should be expected, as the standard error of the true value is smaller than that of the estimate. From the numerical results, it can be said that the `mif` function has recovered a fairly good represents of the original dynamics parameter, `d`.

This ends the demonstration of the `mif` function for the `nLnG` package. Included in Appendix C, a comparison of methods is presented for the reader, showing the

Value	Truth	MIF Estimate
`d`	1.00	0.9993414
`loglik`	-22.18221	-24.42531
`loglik.se`	0.1679821	0.5213355

Table 3.6: This table compares the estimated parameter values for `d`, the log-likelihood value and the standard error to their true values for the oridnary `mif` parameter test using the example animal tracking problem.

equivalence of the methods in package `nLnG`, filter to filter, and smoother to smoother when the data set is linear. One may choose to skip the findings in Appendix C and continue straight to Chapter 4, with the demonstration of package `nLnG` with a real set of observed animal tracking data.

Chapter 4

Observed Data Analysis with nLnG

In this chapter the focus will be illustrating the use of package nLnG while working with a set of real data. The process of creating the SSM for the observed data varies from what has have seen so far in Chapter 3 with the simulated data. Therefore a full exploratory analysis of the data set will be presented to show the difference when working with observed data using the software. The chapter will begin with doing a explanation and visualization of the observed data being used, show the user how to create the SSM in nLnG for the observed data, estimate parameters for the SSM, then use the other package applications to determine the true state of the system (in this case the true position of the animal). Let us first introduce the data for the observed animal track.

4.1 Data Introduction

The observed data set that will be used for this demonstration is a set of Atlantic grey seal data. The observations are recorded using GPS tags placed on the seal and this dataset has been supplied by the Ocean Tracking Network (OTN) research group at Dalhousie University. OTN is a worldwide conservation project currently headquartered at Dalhousie University, Halifax, Nova Scotia and they aim to increase the information available on the world's oceans. There is much to learn about the depths of the ocean waters, particularly their importance in sustaining the quality of animal life around the world. Using acoustic telemetry technology to track thousands

of marine species around the world, from fish to birds, these data can be used to build a library of data to understand animal life in the ocean. OTN has sponsored several studies on the Atlantic Grey seal to determine effects of the species' population on their surrounding environment. Such studies, have been made possible recently with the advancements in technology and tagged tracking implemented by satellite telemetry. By tagging animals, movements over large distances can followed (Roland et al., 1996; Bergman et al., 2000; Block et al., 2001). The tagged animals produce movement pathways from their location and time being recorded from satellite monitoring over a period of time.

These pathways are time-series of location observations of the animal, and can be modelled using SSMs. These pathways have been used in the past to do studies such as meta-analysis of animal movement (Jonsen et al., 2003) and to examine seasonal foraging tactics (Breed et al., 2009). Each of these studies and others have used SSMs in their analyses of animal pathways data. This connection between SSMs and the animal's pathways lead to the idea of the animal pathways being thought of as correlated random walks (CRW) (Kareiva and Shigesada, 1983; Marsh and Jones, 1988; Turchin, 1998; Okubo and Gross, 2002). Even though the idea of using CRW for animal's pathways is an established one, fitting the models to data was difficult (Turchin, 1998; Okubo and Gross, 2002). The fitting of these models has become easier by using MCMC simulations or particle filters (Gelfand and Smith, 1990; Doucet et al., 2001) as compared to earlier analytical or numerical methods (Kalman, 1960). Particle filters are an SMC method and these type of methods even offer benefits over MCMC such as real time calculation, computational efficiency, and straightforward implementation of NLNG models (Breed et al., 2009). Only a few studies have chosen to use SMC methods to solve ecological problems (Ionides et al., 2006; Dowd and Joy, 2011), though both these particle filter and smoother methods

have been shown to be used for state estimation for animal pathways (Royer et al., 2005; Andersen et al., 2007) . In this demonstration SMC methods will be heavily used as the package `nLnG`'s NLNG methods are all of this type. Modelling choices will be discussed more in the next section.

OTN has supplied two sets of observed Atlantic grey seal data for this thesis from one of their studies. The first animal's observed pathway can be seen in Figure 1.1 in Chapter 1 and while the second animal's pathway can be seen in Figure 4.1. The second pathway in Figure 4.1 will be used for this demonstration. It was recorded from July, 14, 2011 till December 28, 2011 (the section used for the demonstration is from July, 14, 2011 till July 16, 2011). In Figure 4.1, the red line shows that the pathway starts travelling on land from the west side of Sable Island area to the east side of the island, then moves southeast out into the water and after a period starts swimming back northeast. This transition from land to water also corresponds with a significant jump in time between observations being recorded.

In Listing A.26, a sample of the recorded data for one of the animals can be seen. The sample of the recorded data shows the data includes an identifier (EID), the animal's tag number (this one was 106709 and the file name for the data was 106709f, where the f indicates the seal was a female), the date and time of the observation and the latitude and longitude positions at that time for each observation made. The other information included in the data is the speed the animal was moving at the time of the observation, the amount of time (seconds) between observations and the cumulative time in seconds (which has been added to the data). Now that the real observed data for demonstrating the use of the package `nLnG` has been introduced, let us now move into the modelling process for the SSM that will be used to estimate the true state (unobserved state) of the system.

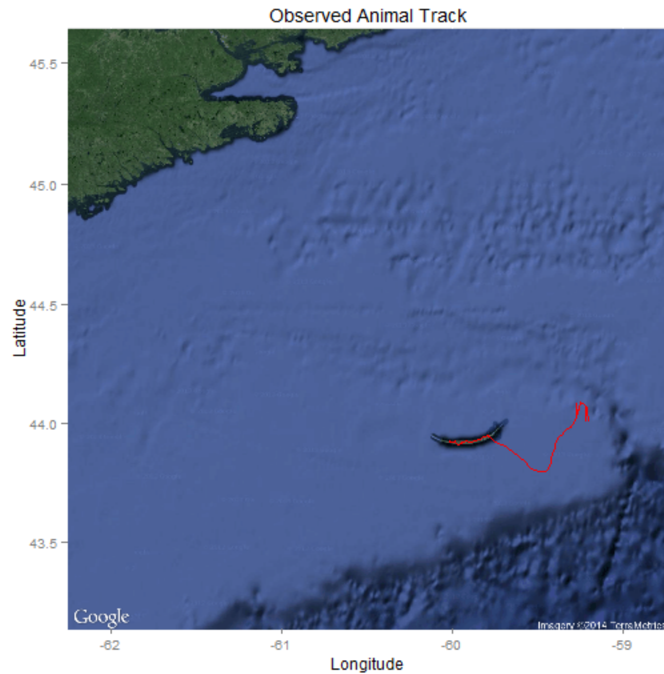


Figure 4.1: Plot of the Atlantic grey Seal track from the Scotian Shelf that will be used to demonstrating the use of nLnG with real data.

4.2 Model Selection and Parametrization

4.2.1 Model Selection

The process of creating a SSM for an observed data set is much different than what has been seen so far in Chapter 3 with the simulated animal pathway. In the case with simulated data, the observed state and the true state for the animal pathway were both known, so it was a little bit like cheating in terms of the modelling process. When working with observed data, one generally does not have the true state of the system and that is what is generally being recovered from the observed data.

The process of creating a SSM for the observed data can be long and tedious, but it is good practice to look to past research into the data to get help. With the observed Atlantic grey Seal data there is a considerable past research, some of which was presented in either Section 1.1 or Section 4.1. Most of this research used some type of SSM to mimic the movement for the state equation. The latest research used a CRW to represent the state equation (Breed et al., 2012). This model is similar to the single state first-difference CRW used by Jonsen et al. (2005) as the state (process) equation and will be the model used here to try to fit to the observed data. The CRW model is described by the following set of equations:

$$\mathbf{d}_t = \gamma_t \mathbf{T} \mathbf{d}_{t-1} + \mathbf{v}_t, \mathbf{v}_t \sim N_2(\mathbf{0}, \Sigma_t), \tag{4.1}$$

$$\mathbf{d}_t = \mathbf{x}_t - \mathbf{x}_{t-1}, \tag{4.2}$$

$$T = \begin{pmatrix} cos\varphi_t & -sin\varphi_t \\ -sin\varphi_t & -cos\varphi_t \end{pmatrix}, \tag{4.3}$$

$$\Sigma = \begin{pmatrix} \sigma^2_{lon,t} & 0 \\ 0 & \sigma^2_{lat,t} \end{pmatrix}. \tag{4.4}$$

$$\varphi_t \sim wC(0, c_t) \tag{4.5}$$

The displacement between unobserved states, $\mathbf{d}_t$, is described in Eq. 4.2 as the difference between the positions $\mathbf{x}_t$ and $\mathbf{x}_{t-1}$. γ_t represents the correlation magnitude and direction of consecutive displacements. The turn matrix, T, is described in Eq. 4.3 with *cos* and *sin* on the cross diagonals and each has the turn angle parameter, φ_t, associated with it. The parameter is then described in Eq. 4.5 by a wrapped

Cauchy distribution (results from the "wrapping" of the Cauchy distribution around the unit circle) centered at 0 and with a concentration parameter, c_t, for the error term. In Eq. 4.1, the correlation term, γ_t, is applied to T and is multiplied by the displacement at $t-1$, $\mathbf{d}_{t-1}$. The error term is then described as a bivariate Normal distribution with mean, 0, and variance-covariance matrix, Σ_t. Σ_t is diagonal with entries $\sigma^2_{lon,t}$ and $\sigma^2_{lat,t}$, the error terms for the change of location position.

Eq. 4.1 can then be rewritten in term of $\mathbf{x}_t$ to fit into the SSM form that has been represented so far in this thesis, by adding $\mathbf{x}_{t-1}$ to each side and substituting Eq. 4.2 in for $\mathbf{d}_{t-1}$. This yields the following equation:

$$\begin{aligned}\mathbf{x}_t = \mathbf{x}_{t-1} + \gamma_t \begin{pmatrix} cos\varphi_t & -sin\varphi_t \\ -sin\varphi_t & -cos\varphi_t \end{pmatrix} (\mathbf{x}_{t-1} - \mathbf{x}_{t-2}) + \mathbf{v}_t, \\ \mathbf{v}_t \sim N_2\left(\mathbf{0}, \begin{bmatrix} \sigma^2_{lon,t} & 0 \\ 0 & \sigma^2_{lat,t} \end{bmatrix}\right).\end{aligned} \tag{4.6}$$

Lastly, this is not in markovian form since $\mathbf{x}_t$ cannot depend on both $\mathbf{x}_{t-1}$ and $\mathbf{x}_{t-2}$. To fix this define $\mathbf{x}_{t-1} = \mathbf{z}_t$ and the augmented state as:

$$\dot{\mathbf{x}}_t = \begin{bmatrix} \mathbf{x}_t \\ \mathbf{z}_t \end{bmatrix}.$$

Eq. 4.6 can now be expressed in markovian form in terms of $\dot{\mathbf{x}}_t$.

In comparison to the example data SSM used in Chapter 3, the main differences between the models are there is no inclusion of a drift term here and the addition of the correlation term based around the displacement between the last two time steps, $\mathbf{x}_{t-1}$ and $\mathbf{x}_{t-2}$. This will be the form of the Breed et al. (2012) SSM that will be used for the state equation for this test. With the state equation defined, let us move to the observation equation.

The observation equation that relates the unobserved location, $\mathbf{x}_t$, to the observed position, $\mathbf{y}_t$, for the Breed et al. (2012) model is described as follows:

$$\mathbf{y}_t = \dot{\mathbf{x}}_t + \mathbf{e}_t, \mathbf{e}_t \sim N_2\left(\mathbf{0}, \begin{bmatrix} \sigma_\eta^2 & 0 \\ 0 & \sigma_\eta^2 \end{bmatrix}\right). \tag{4.7}$$

The model in Eq. 4.7 closely resembles the observation model from the example SSM used in Chapter 3. The observation, $\mathbf{y}_t$, again depends only on the state, $\mathbf{x}_t$, and an error term, where σ_η^2, is the variance of the latitude and longitude found in Costa et al. (2010a) and represents the measurement location uncertainty. Now that the SSM to apply to the observed Atlantic grey seal movement track to has been defined, let us move into parametrizing the model based on the new data.

4.2.2 Parametrization

The parametrization will consist of specifying a group of parameters. The non time-varying parameters are made up of just the error term for the equipment reading for latitude and longitude positions in the observation model, σ_η^2, where:

$$\sigma_\eta^2 = 0.036\, km^2. \tag{4.8}$$

The standard error term in the process model for the longitude position, σ_{lon}, and the process standard error term for the latitude position, σ_{lat}, are:

$$\sigma_{lon} = \sigma_{lat} = 1\, km. \tag{4.9}$$

The time-varying parameters for the state model that need to be estimated include the correlation magnitude, γ_t, and the standard error term for the turn angle

parameter, c_t. So the parameter space for the CRW model can be represented in the following manner in terms of θ:

$$\theta_t = (\gamma_t, c_t) \tag{4.10}$$

With the parameter space now defined, it is now necessary to give initial starting points for each of the parameters for the estimation. Let us look at the values selected for each of the parameters in θ_t.

In Breed et al. (2012), the parameters γ_t and c_t range from 0 to 1. During the parameter estimation procedure a few special methods were used. There were transformations used on the parameters so that they can be added as Gaussian noise in the augmentation part of the estimation; a mixed-Normal distribution was used in order to allow for rapid parameter value variations to deal with animal behaviour changes. This experiment will only deal with recovering the travel behaviour state and therefore will not require the use of a mixed-normal distribution for the behaviour changes, so only one half of the model parameters used in Breed et al. (2012) will need to be specified. In the Simulation Section of Breed et al. (2012), listed under Design 2; are a set of fixed parameters used for the switching model parameters in one of the simulations performed. The set of fixed parameter values used for the travel behaviour state are listed as $\gamma_1 = 0.9$, $c_1 = 0.98$, and $\sigma_{lon,1} = \sigma_{lat,1} = 1$ km (defined above) (Breed et al., 2012). These point values from the simulation will act as good initial parameters guesses for performing parameter estimation with the observed data for this demonstration. The parameters for the observed data will likely not be equal to these, but they are good parameters to begin the process with.

Now that the model and parameters chosen to mimic the animal's movement have been explained, with the initial values of the parameters from Breed et al. (2012) acting as entry points, the parameters for the unobserved data set can be now estimated to fit the model.

4.3 Parameter Estimation

In this section the observed data set will be used to estimate the model parameters. These will be then used to simulate movement tracks to perform filtering and smoothing upon. First, the observed data will be encased into a `nLnG` object for the reader and the highlights explained, before using the created object for the parameter estimation process.

4.3.1 Preparing the Observed Data `nLnG` Object

The job of creating the `nLnG` object is similar to what was seen in Chapter 3, with a few small changes. The object needs to have the appropriate parts defined for running the specific procedure, the initial time, the initial parameter vector and the data frame. The data frame is where the biggest change occurs from what was seen in Chapter 3. Instead of just setting up the names for the data and filling them from the simulation, the observation and their times need to be included in the data frame. Let us examine the `nLnG` object now for the observed data.

Listing 4.1 shows the creation of the `nLnG` object for the observed data. Looking at the object's function call, all the parts of the object for performing MIF parameter estimation have been defined. The `realtrack.ex` object contains a data frame (which will be discussed below), a set of times, a initial time, a state simulation function, a observation density function and a parameter vector. The biggest change from what was seen in Chapter 3 is in the `data.frame`, as it now contains the set of times, longitude and latitude positions from the observed data (`data2` here). The data is set up from the 3rd record to the 53rd to grab the first set of 50 observations, with the

first two positions kept as initial conditions for the first state transition. The `times` attribute is set to be a pointer to the time specified in the `data.frame` and the initial time position, `tInit`, is set to the 2nd time-step in the time vector (`data2$X[2]`). The object has 2 functions filled in for performing state transitions and calculating observation densities. The `state.sim` function is using the `discrete.time.sim` wrapper as the method to advance from one time to the next (this selection will be

```
realtrack.ex<-nLnG(data=data.frame(time=as.numeric(
                                          data2$X[3:53]),
                                     ylon=data2$lon[3:53],
                                     ylat=data2$lat[3:53]
                                     ),
                   times="time",tInit=data2$X[2],
                   state.sim=discrete.time.sim
                   (step.fun=real.track.state.sim,delta=1),
                   obsev.den=real.track.obsev.den,
                   parms=theta)
```

Listing 4.1: Creating the `nLnG` object for the unobserved Atlantic Grey seal data to be used for the parameter estimation test.

discussed further later). Lastly, the initial parameter vector, `theta` (introduced in the next section), is specified to complete the `nLnG` object for the observed data. Let us describe the included functions now.

In Listing A.27, the function for the state transition (an implementation of Eq. 4.6) can be seen. Looking at the function, the 2 time-varying parameters, `gam` and `c`, have some type of limit set on their values. The angle for the turn matrix (T) is

produced from a wrapped Cauchy distribution (using package CircStats (Lund and Agostinell, 2018)). The new state position (`xnew`) is calculated from a multivariate Normal distribution, then checked to make sure it is within a certain area. The issue was at times a few of the particles were taking on odd values. This caused the `mif` procedure to fail, so the large box catches those values and replaces them with a value that will not cause the filter to fail or have the state hold any weight in the measurement part of the filtering. The new state, `xnew`, is then combined with the previous x_t position, renamed with the correct names and returned to the procedure.

In Listing A.28, the implementation of Eq. 4.7 for the observation density model can be seen. The function is straightforward, as the observation error, `nada`, is pulled from the parameter vector and used to compute the conditional likelihood of state, $\mathbf{x}_t$, given the observation, $\mathbf{y}_t$, by multivariate Normal distribution. The conditional likelihood is then returned to the procedure. Now that the object has been created to perform the parameter estimation, let us move to performing the test for the observed Atlantic grey seal data.

4.3.2 Simulation Testing

During this parameter estimation test, the 2 time-varying parameters in Eq. 4.10 will be estimated using the `mif` application included in package `nLnG`. Let us start by introducing the initial parameter vector for the `nLnG` object in Listing 4.1. The initial parameter vector, `theta`, holds the starting positions for the first state transition and the initial parameter value guesses for the estimation procedure. Table 4.1 links the parameter vector in Listing 4.2 to the SSM in Eq. 4.6 and 4.7. Table 4.1 shows the

```
theta <-c(c=0.98,gam=0.9,xlon1.0=-60.01687,xlat1.0=43.92794,
          xlon.0=-60.01379,xlat.0=43.92359,slon=0.539957,
          slat=0.539957,nada=0.1)
```

Listing 4.2: Initial Parameter Vector.

the observation model, `nada` and it has been set to the static value from Costa et al. (2010a). With all the initial values introduced for the parameter vector, `parms`, for the unobserved data object, let us move now to setting up the test for the parameter estimation.

Mathematical Symbol	`nLnG` Variable	Value
γ_t	`gam`	0.9
c_t	`c`	0.98
$\sigma_{t,lon}$	`slon`	0.539957
$\sigma_{t,lat}$	`slat`	0.539957
$\mathbf{x}_{t,lon}$	`xlon.0`	-60.01379
$\mathbf{x}_{t,lat}$	`xlat.0`	43.92359
$\mathbf{x}_{t-1,lon}$	`xlon1.0`	-60.01687
$\mathbf{x}_{t-1,lat}$	`xlat1.0`	43.92794
σ^2_η	`nada`	0.1

Table 4.1: This table shows the link between the mathematical symbol in Eq. 4.6 and 4.7, the variables supplied for use in Listing 4.2 and the initial value for the parameter.

The parameter estimation test using the `mif` application can be viewed in Listing 4.3, starting with the set up for the test and then the calling of the `mif` function. Referring to Listing 4.3, the 2 time-varying parameters names are first placed in the vector, `parEst`. Next, the test is set to be replicated, though is currently only being

run the one instance. The estimation parameter vector, `guess`, is given the parameter vector, `theta`, in Listing 4.2 and then the starting position for the parameter values. The parameters γ and c_t are set using a uniform distribution to get the values between 2 time-varying parameters initial values are set at the prior values from Breed et al. (2012) (the error for the state position has been converted from kilometers to nautical miles). Next are the initial longitude and latitude positions for states $\mathbf{x}_t$ (`xlon.0` and `xlat.0`) and $\mathbf{x}_{t-1}$ (`xlon1.0` and `xlat1.0`). The first two observations of the data set are used for the initial points, this also mean that both $\mathbf{x}_t$ and $\mathbf{x}_{t-1}$ will be stored in the `nLnG` object. $\mathbf{x}_{t-1}$ is stored for convenience of it being easier to access for the state model in Listing A.27. Last in Table 4.1 is the error term for the desired interval of 0 and 1. With the starting points for the parameters being estimated now set, next we move on to the function call for the `mif` application.

The function call for the `mif` is fairly straightforward and the key attributes will be highlighted here. The function is supplied the `nLnG` object, `realtrack.ex`, containing the parts of the SSM from Listing 4.1. The number of mif iterations, `Nmif`, is set at 50. The starting point for the parameters, `start`, is set to the vector, `guess` (which gets the initial points for the estimation during the `mif` setup). The vector of parameter names to estimate, `pars`, is defined. Also set are the error value for the parameters being estimated, `rw.sd`, and the number of particles used, `Np`, is set to 2000. The variance factor, `var.factor`, is set to be 4 times to begin and the variance cooling, `cooling.factor`, set to 0.99. This finishes the setup for the parameter estimation for the `mif` test. Next we will look at the results from the procedure.

```
###parameters to estimate
parEst <- c("c","gam")
###mif test
realTrack.mif <-replicate(
  n=1,
  {guess <- theta
  ##giving differnt starting point to the parameter
  guess[("c")] <- runif(1,0,1)
  guess[("gam")] <- runif(1,0,1)
  ##call to the mif function
  mif(
    realtrack.ex, Nmif=50,
    start=guess, pars=parEst,
    rw.sd=c(c=0.075,gam=0.2),
    Np=2000, var.factor=4,
    ic.lag=51, cooling.factor=0.99,
    max.fail=500, verbose=TRUE)
  })
```

Listing 4.3: Observed data parameter estimation test, running the `mif` function for determine the 2 time-varying parameters, `gam` and `c`.

Figures 4.2 and 4.3 shows the results from using the the `mif.comp` function in the package. Figure 4.2 are the diagnostic plots for the last iteration of the filter and Figure 4.3 are the diagnostic plots of the MIF convergence.

Figure 4.2, the last filter iterations results shows plots of the effective sample size, `xlon`, `xlat`, `c` and `gam`. An issue can be seen within all the the plots in the Figure,

in that at about the 50000 second mark; there is a large time jump between recorded observations and the next one is not till about 175000 seconds. Referring to the individual plots in the Figure, it can be seen that the values before the time jump tend to be vastly different from the values after the jump in time. This could lead to issues with the estimation process, mainly that the observed track may have

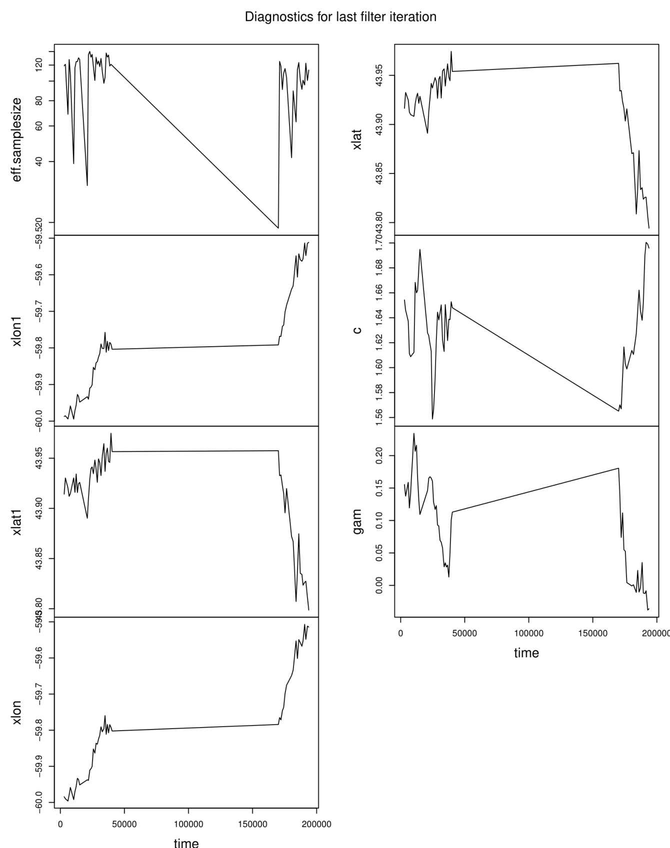


Figure 4.2: Shows the diagnostics plots of the last filter iteration for the parameter estimation test for `c` and `gam` using the real observed animal data. Plots included in the panels are the effective sample size, the estimated states (`xlon1,xlat1` get plotted from inclusion) and the estimated parameters over the time (seconds). Note: the long straight line in the middle represents the jump in time between observations.

contained little information to begin with and with a large time jump in the middle of the track, the information could quite possibly change after the jump (implication further discussed in Conclusion).

Now moving to Figure 4.3 shows the plots of the convergence for the `mif` test. Looking at the 4 panels of the plot, it shows convergence diagnostics are returned for the log-likelihood value (`loglik`), the number of failures (`nfail`), and the two (2) parameters that were estimated during the test (`c` and `gam`). The log-likelihood convergence shows that the value begins around -119 and slowly oscillates in an increasing manner throughout the iterations, with it ending at approximately -116.5. Next `nfail` shows that the test consistently had no failures during the estimation process. Next is the convergence plot panels for the two parameters that were estimated. The convergence plot of `c` shows the value starts at approximately 0.4 and while the trace does not converge constantly, its steadily increasing again throughout the iterations; with it ending at a value slightly above 1.6. Lastly, the convergence plot of `gam` shows that the value begins at approximately 0.2 and wildly oscillates between -0.6 to 0.6 throughout the iterations, ending at value slightly below -0.2. In Breed et al. (2012) `gam` is a value changing from 0 to 1, and this could explain the value oscillating about zero and not appearing to converge during the process. This could be caused by a few issues affecting the process such as parameter trade-off, covariance, and/or the observed data not containing enough information. We will continue on with the analysis with the estimated parameters even though neither parameter appears to converge during the test, but will have a further discussion on the estimation process in the Conclusion. Now that we looked at the plotted results for the `mif`, let us look at the estimated value for the parameters from the test.

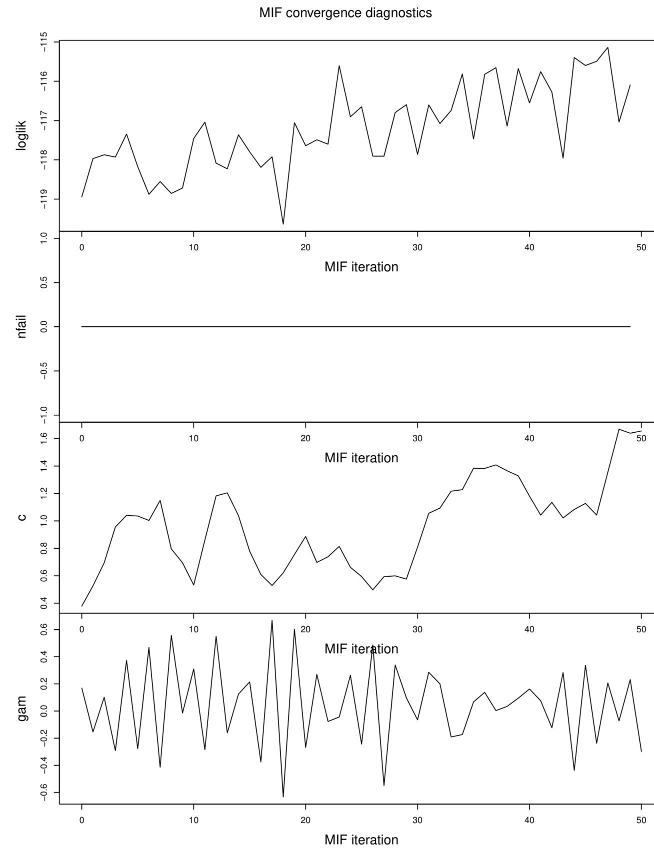


Figure 4.3: Shows the convergence diagnostics plots for the parameter estimation test for `c` and `gam` using the real observed animal data. The 4 panels show plots of the log-likelihood value, number of failures, and estimated parameters values over the course of the `mif` iterations.

In Table 4.2, it shows the values for the estimated parameters from the `mif` test. The estimated value for the `c`, which influences the turning angle, was 1.6559215 and the estimated value for the `gam`, which weights how correlated the next state position is to the previous was -0.2979874. Next are some performance numbers for the estimated values, the average log-likelihood for the test was -114.9911 and the standard error for the log-likelihood was 1.01391. Overall the estimated values seem

to be a little off from what was expected. The value for `c` is above the limit we expected to see and outside the range that was set. While the value for the `gam` parameter is a little lower than the expected range, but not seriously out of question as a plausible value. This issue may have been caused by the problems addressed in the analysis of the `mif` diagnostics plots above. Now that we have estimated new parameters for the state model based on the observed animal track data, let us move on to testing how the parameters perform with the other NLNG applications in `nLnG`.

Value	**MIF Estimate**
`c`	1.6559215
`gam`	-0.2979874
`loglik`	-114.9911
`loglik.se`	1.01391

Table 4.2: This table shows the values from the `mif` test for the two estimated parameters (`c` and `gam`), the log-likelihood value for them and the standard error for the log-likelihood value.

4.4 Filtering and Smoothing Results

In this section the estimated model parameters will be used first to perform particle filtering and then, second, to smooth the result using the particle smoother. Let us start with using the `pFilter` application to obtain the filter states to use for the smoothing process.

4.4.1 Filtering

For the filtering process the particle filter will be used. The user call for the function can be seen in Listing 4.4. The following are supplied: the `nLnG` object, `realtrack.ex`, containing the state and observations function defined in Listing A.27 and A.28, the saved parameter estimate, `theta.mif`, from the `mif` procedure, the number of particles is set at 1000 and we set all the statistical measurements to be collected. Now that we highlighted the test to be conducted, let us examine the results from it. We will start by looking at the effective sample size plot.

```
realTrack.pf <- pFilter(realtrack.ex,parms=theta.mif,
                        Np=1000,pred.mean=TRUE,
                        pred.var=TRUE,filter.mean=TRUE,
                        filter.var=TRUE,save.states=TRUE)
```

Listing 4.4: Call for `pFilter` function for the observed data using the parameters estimated using the `mif` function.

In Figure 4.4 it shows the effective sample sizes for the particle filter test using the real observed animal data. The particle size used for the test was 1000. Figure 4.4 shows the maximum number of unique particles in the sample was approximately around 80 and the smallest number of least unique particles in the sample around 20 (a few times). The number of unique particles in the sample mostly averaged about 60. These effective sample sizes are smaller proportion wise than what was seen with the example problem, but are an effect of now working with more complex models and actually observed data; since the model may not be consistent with the data. Now that we have looked at how the filter performed for the estimated parameters.

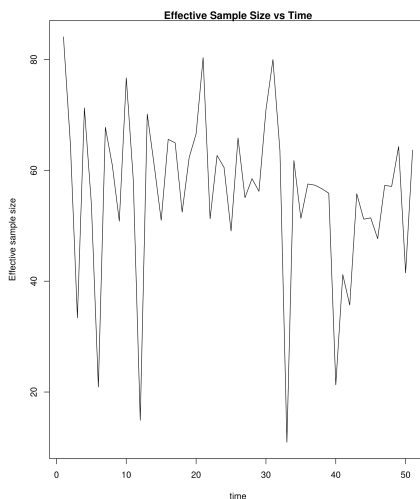


Figure 4.4: Plot of the effective sample size for the `pFilter` result for the real observed animal tracking data (max number of particles was 1000).

Let us next look at how the particle filter did at recovering the states.

In Figure 4.5 the top plot is the 2D plot of the filter states (red) superimposed on the observations (blue), with the smaller two plots being the individual longitude and latitude positions versus time. The 2D plot shows that the observed states (blue) appear to be in the middle of the filter states (red). The variations of the filter states appear to be generally large from the observed states at times, making the filter state appear more jagged. The two individual component plots of the longitude and latitude in Figure 4.5, maybe shows the more interesting finding. In each plot, during the middle of the time period, there is a large jump in time (or a large period of time without an observation), resulting in a long straight line from timestep 30000 to 175000 (this long time jump may also have effects on the parameter estimation process, which will be added to the discussion in the Chapter 5). The beginings and

ends of each of the component plots appear as we would expect without the long straight middles. In the longitude component plot (on the left) the filter states (red) follow fairly closely to the observations (blue), while in the latitude plot (on the right) the filter states (red) are much more varied from the observations (blue). From the results, we can say the estimated parameters did an adequate job at producing filter states for the observed animal track.

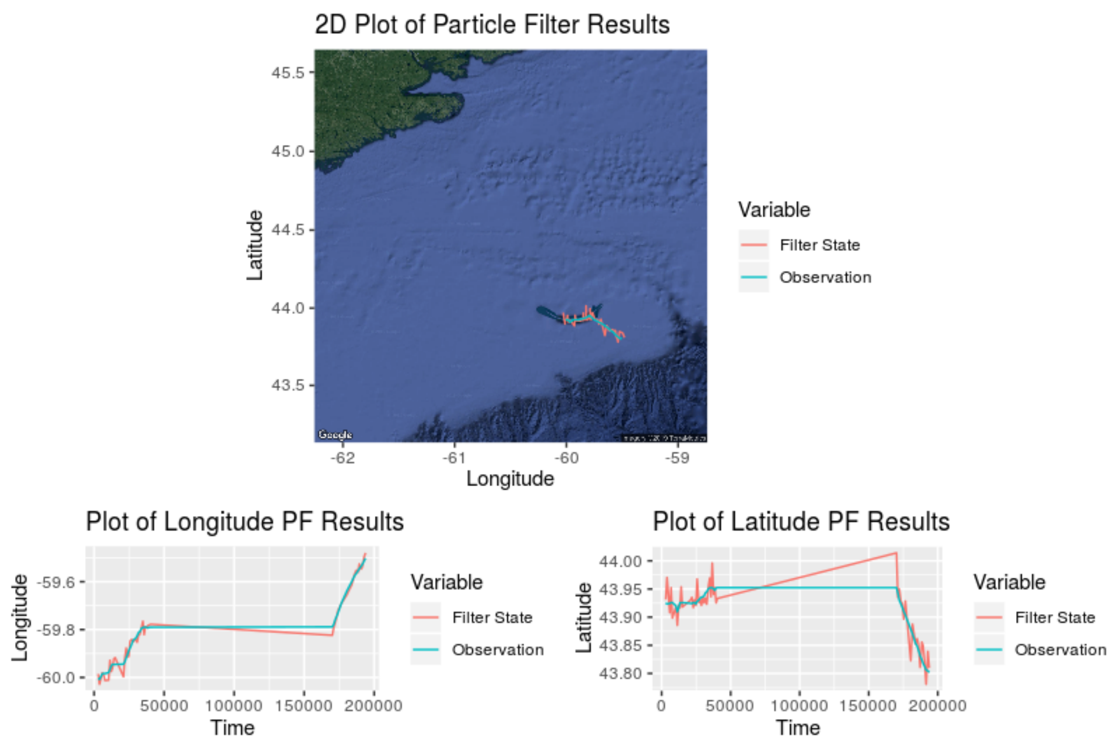


Figure 4.5: Plots of the results for the `pFilter` function. Top: 2D plot of the filter states (red) vs the observations (blue). Bottom: component plots of longitude and latitude variables versus time.

Now that the result for the observed animal data has been filtered and the object, `realTrack.pf`, prepared for using with the particle smoother application in `nLnG`. Let us now smooth the filtered result from above.

4.4.2 Smoothing

In this section, the result from the `pFilter` application for the observed animal tack data will be used to demonstrate the particle smoother application, `pSmoother`, in `nLnG`. In Section 4.4.1 the object, `realTrack.pf`, had the ensemble states saved and prepared for use with the `pSmoother`. Let us first start by describing the set-up of the `pSmoother`.

Listing 4.5, shows the call to the `pSmoother` function for the observed data test. The function call used the `pFilterd-nLnG` object, `realTrack.pf`, from the particle filter test, the estimated parameters from the `mif`, `theta.mif`, number of particles,

```
realTrack.ps <- pSmoother(realTrack.pf,parms=theta.mif,
                          smoother.mean=TRUE,
                          smoother.var=TRUE,
                          Nreal=25,Np=1000,xhat=TRUE,
                          save.real=TRUE,verbose=TRUE)
```

Listing 4.5: Call for `pSmoother` function for the observed data, running 25 realization, with 1000 particles used and collection all the values for smoother mean & variance, the xhats, and the realizations.

`Np`, number of realization, `Nreal` and was told to collect a number of quantities during the process (smoother mean and variance, the $\hat{\mathbf{x}}_t$, and the realizations). With the test performed now highlighted, let us examine the results.

For the smoother, we will use two sets of plots to assess the results from the test. The first set of plots will be the regular 2D and component plots. The second set of plots will be comparison of the particle filter and smoother results. Let us examine

the results of the smoother first.

Figure 4.6, shows the plots of the particle smoother results plotted with the observed data. In the large 2D plot at the top of the Figure, the smoother state (blue) does fairly well at the start of the observations (red) before separating from it. The smoother state does correct itself to the observations before another set of large deviations (this is where the large gap in observations takes place). After the time jump, the smoother state again corrects itself to the observations, then with another slight separation before finishing back on top of the observations. Now referring to the individual components in the lower 2 plots in Figure 4.6. Starting with the longitude

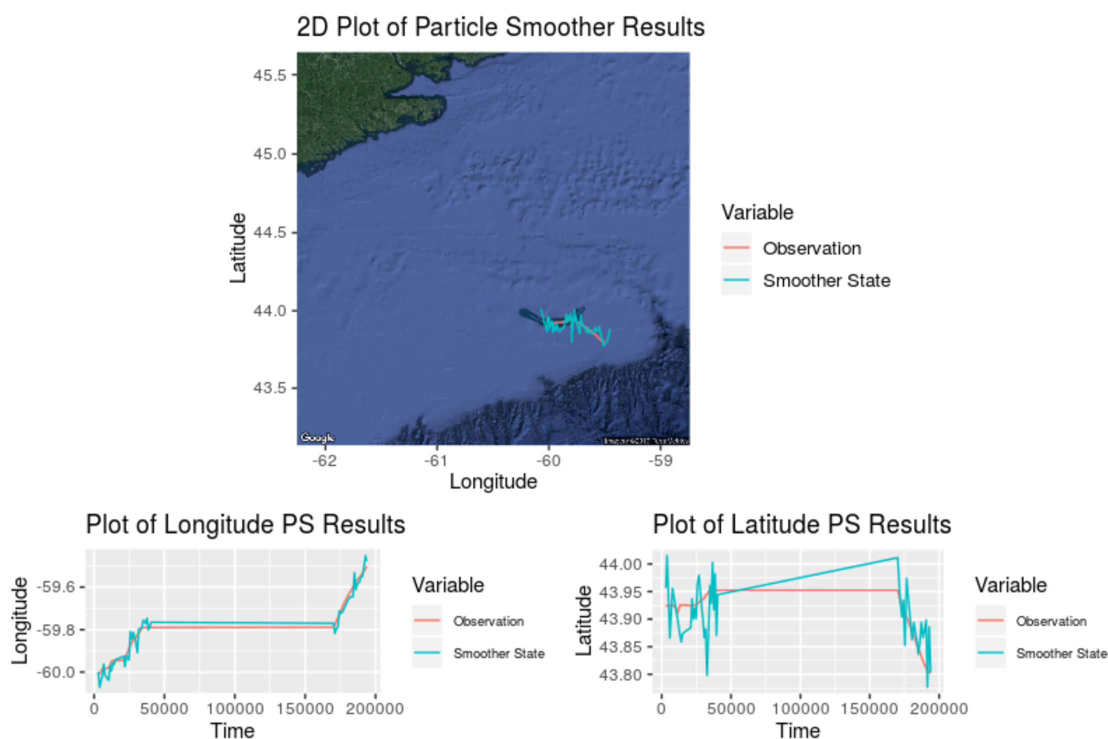


Figure 4.6: Plots of the results for the `pSmoother` function. Top: 2D plot of the smoother states (blue) with the observations (red). Bottom: component plots of longitude and latitude variables versus time (seconds).

component plot (on the left), the smoother state (blue) is fairly close to the observations (red) throughout the plot even after the time jump. Looking at the latitude component plot (on the right) the smoother state (blue) does much worse at following the observations (red) and is barely on top of the observations throughout the plot at all. Next let us look at the smoother state estimated in contrast to the filter state before discussing the results.

Figure 4.7 shows the plots of the smoother state plotted with the filter states and observations. Referring to the large 2D plot at the top, note that the smoother state (blue) only does a good job at smoothing a few of the spots of the filter state (red) (mostly where the smoother estimated state did a good job of following the observed states (green)). Referring to the 2 component plots in the lower part of the Figure. The plot of the longitude (on the left) component shows the smoother state (blue) does a fairly good job of going through the middle of the filter state (red), while in the latitude component plot (on the right) the smoother state (blue) does a very poor job at eliminating the filter state's (red) roughness. Now that we have examined the plots of the particle smoother results and contrasted them against the particle filter results, let us move to discussing the overall performance.

Overall, the particle smoother result for the observed data seems to do only an average job of estimating the true state (observed in this case) of the animal. While it seems that the longitude variable did a much better job of estimating the true state, the latitude variable's terrible performance of estimating the true state is probably a big factor in the overall result. This could be caused by a couple of different reasons. First the number of realizations run for the particle smoother was rather small (only 25). Normally you would want to run a much larger set (discussed further in the Conclusion). Also the lack of information in the observed data for estimating the

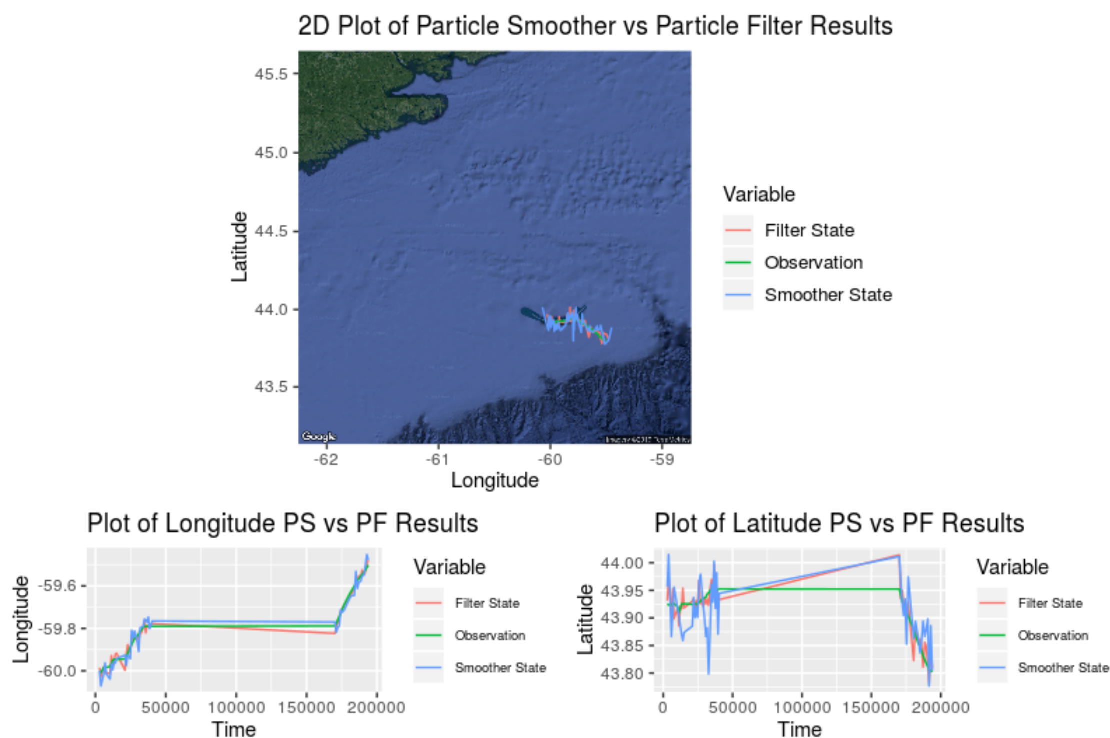


Figure 4.7: Plots of the results for the particle smoother vs particle filter. Top: 2D plot of the smoother states (blue) vs the observations (red). Bottom: component plots of longitude and latitude variables versus time (seconds).

parameters to fit the Breed et al. (2012) SSM likely contributed to the poor performance.

This brings the demonstration of the `nLnG` package with the observed Atlantic grey seal data to an end. Next there will be a discussion in the Conclusion about the overall performance in general, as well as package creation for `R` and remarks on future work for the package.

Chapter 5

Conclusion

In this conclusion we will first do a brief discussion on the creation of the `nLnG` package, followed by a full discussion on the demonstration of the package with the observed Atlantic grey seal data and end by discussing future steps for the package.

5.1 Package Creation

From the package creation aspect of the thesis, it would be easy to say that `nLnG` accomplished what the main purpose of the thesis to develop an R package for NLNG SSMs, with the unique inclusion of a particle smoother. The package itself passes all the CRAN checks to install and has been installed on multiple systems. The package offers a variety of SSM applications of interest, that support both single and multivariate types of model structures, with quick to use diagnostics and comparison plotting to analyse results. The creation of nLnG using R's infrastructure at the beginning was a slightly tedious process, with having to copy files and prepare the package folder with all the appropriate parts for error checking during programming and testing. Switching to using **RStudio** allowed for some of the tedious aspects to be removed, as it has options to create packages as a project, which allows the user to automate some of the error checking process. One of downfalls of using **RStudio** to generate packages was that the none of the `man` pages were generated, which still left the user significant work to prepare this folder. My solution to the issue was to

create a script, that I would run to collect the missing files from the previous version of the package folder and copy them to the new package folder (**RStudio** was very new at the time of the creation of nLnG, and far more documentation/examples are available currently).

One of the main issues that still occurs with nLnG, is that while the analysis of simulated data can be done in a fairly timely fashion, introducing real observed data into the process; meant the analysis process becomes slow and very computational heavy. Even with the inclusion of compiled code to perform much of the computational heavy mathematics, R's overhead system cost really becomes a burden to the execution speed of the programs when introduced with real observed data to analyse (This of course is based on one example data set tested). Let us now move to discussing the analysis of the observed Atlantic grey seal data in Chapter 4.

5.2 Observed Data Analysis

The analysis of the observed data in Chapter 4, while a good demonstration of using real observed data with package nLnG, did not do a great job of mimicking the observed state of the animal. The poor performance is more based on the modelling process than that of the software. The main contribution of the software to the issue is the time in which it requires to perform the analysis, limiting the possibilities to explore different modelling techniques. With more time during the modelling and parametrization process different time-stepping wrappers could have been explored to test for advancing the state (process) model by either using one time step or Euler steps wrappers instead of the discrete time wrapper. Next with the parametrization

of the model to the observed data, there are a few issues affecting the process. To start, as mentioned in Section 4.3, the observed animal data may simply not contain enough information to fit the model. The parameters being estimated could either have been trading off or had a strong covariance between them. While with more time to perform the analysis, single parameters estimation tests could have been attempted, while holding the other parameters static for the test (we estimated two parameters, which could lead to the issues mentioned above). Also, with more testing time, different bounds from those from Breed et al. (2012) on the estimated parameters could have been attempted for the MIF procedure. Finally, the large time jump in the middle of the observations corresponding to the transition from land to water could be another reason that the observed data lacks information for the estimation process. This transition could also represent a change in behaviour from near shore active to foraging, which would cause a change in the parameter values being estimated during the test. This could easily be the cause of why the parameters do not appear to converge during the MIF test in Section 4.3.

The resulting estimated parameters lead to the rather poor estimates of the true state of the animal for the particle filter and smoother applications. With more analysis time, extra testing could have been performed to explore getting better estimated parameter results for the following applications. Now that we have discussed the demonstrated observed data analysis, let us finish with a small discussion on the future work for the package.

5.3 Future Work

The future of the nLnG package after this thesis can take one of a couple different of paths depending on which I prefer to follow through on. Option one would be to release and maintain the package myself through the CRAN site (or another one of the R package distribution sites). While, this would give me full autonomy over the package going forward, it could also mean any inquiries or package updates (streamline the Kalman application better, add to the parameter estimation methods offered, continue adding on the diagnostic features available, etc) would have to be done fully by me, which could be very time consuming for one person. The other option for the package would be for me to become part of a group of contributors to a current package (ideal target is POMP), where my responsibilities would be to maintain the particle smoother application within the larger scope of things and only need to attend to inquiries and updates for a much smaller portion of the overall package. As well, the particle smoother would have to be throughly tested, and its computational performance improved. This could be done with different process models, using both synthetic data generated under various statistical assumptions as well as real data (that may not be fully consistent with the process model proposed). This concludes the introduction of the nLnG package for this thesis.

Appendix A

Code Appendix

```
setClass('nLnG',
         representation(
            data = 'array',times = 'numeric',
            tInit = 'numeric',state.sim = 'function',
            state.den = 'function',obsev.den = 'nLnG.fun',
            obsev.sim = 'nLnG.fun',initializer = 'function',
            state.update = 'function',
            smoother.den = 'nLnG.fun',
            states = 'array',parms = 'numeric',
            covar = 'matrix',tcovar = 'numeric',
            obsnames = 'character',
            statenames = 'character',
            paramnames = 'character',
            covarnames = 'character',
            zeronames = 'character',
            PACKAGE = 'character',userdata = 'list')
         )
```

Listing A.1: `nLnG` Class Representation

```
useDynLib( nLnG,
           get_nLnG_fun,
           lookup_in_table,
           do_obsev_sim,
           do_obsev_den,
           do_state_sim,
           do_state_den,
           do_init_state,
           do_state_update,
           euler_model_simulator,
           euler_model_density,
           simulation_computations,
           R_Euler_Multinom,
           D_Euler_Multinom,
           R_GammaWN,
           systematic_resampling,
           pFilter_computations,
           systematic_resampling2,
           pSmoother_computations
     )
importFrom(graphics,plot)
importFrom(stats,simulate,time,coef,logLik,window)
importFrom(mvtnorm,dmvnorm,rmvnorm)
importFrom(subplex,subplex)
importFrom(deSolve,ode)
```

```
exportPattern("^[[:alpha:]]+")
exportMethods(
          "$",
          "coef",
          "coef<-",
          "coerce",
          "cond.logLik",
          "continue",
          "conv.rec",
          "data.array",
          "dprior",
          "eff.sample.size",
          "filter.mean",
          "filter.var",
          "init.state",
          "kFilter",
          "kSmoother",
          "logLik",
          "mif",
          "nLnG",
          "obs",
          "obsev.den",
          "obsev.error",
          "obsev.sim",
          "parms.rec",
```

```
"pars.mean",
"pars.var",
"particles",
"pFilter",
"plot",
"pmcmc",
"pred.mean",
"pred.var",
"print",
"pSmoother",
"show",
"simulate",
"smoother.mean",
"smoother.var",
"state.den",
"state.error",
"state.sim",
"state.update",
"stateAugmentation",
"states",
"summary",
"time",
"time<-",
"timezero",
"timezero<-",
```

```
        "window",
        "xhat"
    )
exportClasses(
        "kFilterd.nLnG",
        "kSmootherd.nLnG",
        "mif",
        "nLnG",
        "nLnG.fun",
        "pFilterd.nLnG",
        "pmcmc",
        "pSmootherd.nLnG",
        "stateAugmentated.nLnG"
    )
```

Listing A.2: `NAMESPACE` file for class `nLnG`

```
Package: nLnG
Type: Package
Title: NonLinear NonGaussian State Space Model Tools
Version: 1.0
Date: 2012-08-26
Author: Joey Hartling
Maintainer: Joey Hartling <joeyh@mathstat.dal.ca>
Description:
 Package for non Linear non Gaussian state space model
filtering and smoothering tools
Depends: R(>= 2.14.1),methods,stats,graphics,mvtnorm,
         subplex,deSolve
License: GPL (>= 2)
LazyLoad: TRUE
LazyData: False
BuildVignettes: no
Collate: nLnG-help.R eulermultinom-nLnG.R nLnG-fun.R
        nLnG.R nLnG-methods.R obsev.den-nLnG.R
        obsev.sim-nLnG.R state.den-nLnG.R state.sim-nLnG.R
        state.update-nLnG.R smoother.den-nLnG.R
        init.state-nLnG.R simulate-nLnG.R
        pFilter-nLnG.R pFilter.methods-nLnG.R
        filter.diag-nLnG.R kFilter-nLnG.R
        kFilter.methods-nLnG.R mif.class-nLnG.R
        particles-nLnG.R mif-nLnG.R mif.methods-nLnG.R
        mif.comp-nLnG.R pDesign-nLnG.R pmcmc-nLnG.R
        pmcmc.methods-nLnG.R pmcmc.comp-nLnG.R
        kSmoother-nLnG.R kSmoother.methods-nLnG.R
        plot-nLnG.R stateAugmentation2-nLnG.R
        stateAugmentation.methods-nLnG.R
        pSmoother-nLnG.R pSmoother.methods-nLnG.R
        smoother.diag-nLnG.R compare.diag-nLnG.R
```

Listing A.3: The description file for package nLnG

```
effSampleTime(track.pf)
```

Listing A.4: `effSampleTime` plot function call for the Particle filtered object, `track.pf`, to return the plot of the effective particle sample size of time

```
pm <- pred.mean(track.pf)
pv <- pred.var(track.pf)
fm <- filter.mean(track.pf)
fv <- filter.var(track.pf)
```

Listing A.5: Methods for extracting the predictive and filter means and variances for the `pFilter` function

```
filter.diag(track.pf,2,type="particle",Dimplot=TRUE)
```

Listing A.6: Filter diagnostic call for the particle filtered object `track.pf`

```
##adding state update to model
track.ex <- nLnG(track.ex, state.update=
          onestep.sim(step.fun=track.update)
          )
##setting up error matrices for kalman filter
```

```
oError <- matrix(c(0.7^2,0,0,0.7^2),nrow=2,ncol=2)
sError <- matrix(c(0.4^2,0,0,0.2^2),nrow=2,ncol=2)
```

Listing A.7: Setup for `kFilter` function inputs

```
filter.diag(track.kf,2,type="Kalman",Dimplot=TRUE)
```

Listing A.8: Filter diagnostic call for `kFilter`

```
track.ex <- nLnG(track.ex, obsev.den=track.obsev.den)
```

Listing A.9: Adding `obsev.den` to `track.ex`

```
##Methods for Kalman Smoother
##get smoother mean and var
sm <- smoother.mean(track.ks)
sv <- smoother.var(track.ks)
```

Listing A.10: Showing methods to extract the smoother mean and variance for the Kalman smoothed object, `track.ks`. Storing the extracted information in variables `sm` and `sv`.

```
smoother.diag(track.ks,2,type="Kalman",Dimplot=TRUE)
```

Listing A.11: Calling the `smooth.diag` diagnostic function on the example tracking data for the example tracking problem object, `track.ks`

```
track.pf <- nLnG(track.pf,smoother.den=track.obsev.den)
```

Listing A.12: Adding `smoother.den` function to the `nLnG` object, `track.pf`, making the smoothing denisty calculation available to use

```
smoother.diag(track.ps,2,type="particle",Dimplot=TRUE)
```

Listing A.13: Call to smoother diagnostic function `smoother.diag` for the example tracking problem object, `track.ks`

```
compare.diag(track.pf,track.kf,2,Dimplot=TRUE)
```

Listing A.14: Comparison function diagnostic `compare.diag` call for the comparison of the filter methods in package `nLnG`. `pFilter` vs. `kFilter` for the example tracking problem

```
compare.diag(track.pf,track.ps,2,Dimplot=TRUE)
```

Listing A.15: Calling `compare.diag` on the Particle filter and smoother objects

```
compare.diag(track.kf,track.ks,2,Dimplot=TRUE)
```

Listing A.16: Calling the `compare.diag` diagnostic function on the example tracking data

```
##showing the log-likelihood for MLE section of thesis
##running pFilter over different values of dynamic parameter
dvec <- seq(0.997,1.003,by=0.001)
loglikest <- NULL
for(i in 1:7) {
  theta["d"] <- dvec[i]
  track.mle <- pFilter(
```

```
    track.ex,
    parms=theta,
    Np=1000,
    pred.mean=FALSE,
    pred.var=FALSE,
    filter.mean=FALSE,
    filter.var=FALSE,
    save.states=FALSE
  )
  loglikest[i] <- logLik(track.mle)
}
#####Changing Theta over to the Kalman Filter
###verison parameter d = A, h = Q
theta <-c(A=1.00,Q=1.00,drift=0.05,Xlon.0=-60.5,Xlat.0=44.0,
          slon=0.15,slat=0.075,sOb=0.25)
##running the kFilter for different values of
##dynamic parameter
loglikestKF <- NULL
for (i in 1:7){
  theta["A"] <- dvec[i]
  #running the kalman filter
  track.mle <- kFilter(track.ex,parms=theta,
                       obsev.error=oError,
                       state.error=sError,
                       delta.t=1,pred.mean=TRUE,
```

```
                              pred.var=TRUE,filter.mean=TRUE,
                              filter.var=TRUE
                 )
  loglikestKF[i] <- logLik(track.mle)
}
```

Listing A.17: Computing the Log-likelihood for the dynamic parameter `dvec` using the particle filter function `pFilter` and Kalman filter function `kFilter`

```
> logLik(track.sa)
[1] -42.93537
```

Listing A.18: Computing the Log-likelihood for the returned object `track.sa` from the `stateAugmentation` function, for the dynamics parameter estimation test

```
##setting up and
##running estimation by mif
truth<- coef(track.pf)
##picking parameters to estimate
parEst <- c("d")
```

Listing A.19: Setting up the initial parameter estimates and the picking which parameter we want to estimate for the `mif` function

```
track.Pmif<-replicate(n=10,
                     {guess <- truth
                      ##giving differnt starting point
                      guess[parEst]<-rnorm(n=length(parEst),
```

```
                                mean=guess[parEst],
                                sd=.005)
                 ##call to the mif function
                 mif(track.pf,
                     Nmif=50,
                     start=guess,
                     pars=parEst,
                     rw.sd=c(d=0.001),
                     Np=1000,
                     var.factor=2,
                     ic.lag=50,
                     cooling.factor=0.99,
                     max.fail=100)
                })
```

Listing A.20: Running a MIF for parameter estimation using `pars`.

```
track.Imif<-replicate(n=10,
                     {guess <- truth
                      ##giving differnt starting point
                      point to the parameter
                      guess[parEst]<-rnorm(n=length(parEst),
                                           mean=guess[parEst],
                                           sd=.005)
                      ##call to the mif function
                      mif(track.pf,
```

```
                    Nmif=50,
                    start=guess,
                    ivps=parEst,
                    rw.sd=c(d=0.001),
                    Np=1000,
                    var.factor=2,
                    ic.lag=50,
                    cooling.factor=0.99,
                    max.fail=100)
})
```

Listing A.21: Setting up the parameter vector `guess` varying the parameters for estimate `parEst` (the dynamics parameter in this case) the for each run of the `mif` storing the results for each mif results in `track.Imif`.

```
mif.comp(track.Imif)
```

Listing A.22: Call to use the visual analysis tool `mif.comp` for `mif` object in package `nLnG` for the example tacking problem `track.Imif`

```
track.mifP[[1]]@conv.rec
```

Listing A.23: Example of Extracting the convergence records for the first mif replication from the saved results in `mif` object, `track.mif.pars`

```
> theta
      d      h  drift  Xlon.0  Xlat.0   slon  slat   sOb
```

```
  1.000 1.000  0.050 -60.500  44.000  0.150  0.075 0.250
> theta.mif <- apply(sapply(track.mif.par,coef),1,mean)
> theta.mif
          d         h      drift     Xlon.0     Xlat.0
                 slon       slat        sOb
  0.9993414 1.0000000 0.0500000 -60.5000000 44.0000000
             0.1500000  0.0750000  0.2500000
> loglik.mif <- replicate(n=10,logLik
                         (pFilter(track.mif.par[[1]],
+                                 parms=theta.mif,Np=1000)))
> bl <- mean(loglik.mif)
> bl
[1] -24.54787
> loglik.mif.est <- bl + log(mean(exp(loglik.mif-bl)))
> loglik.mif.est
[1] -24.42531
> loglik.mif.se <- sd(exp(loglik.mif-bl))/
+     exp(loglik.mif.est-bl)
> loglik.mif.se
[1] 0.5213355
> theta.true <- coef((track.ex))
> theta.true
      d    h   drift   Xlon.0   Xlat.   slon   slat    sOb
  1.000   1.00   0.050 -60.500   44.000 0.150 0.07   0.250
> loglik.true <- replicate(n=10,logLik(
```

```
                                    pFilter(track.ex,
+                                           parms=theta.true,
+                                           Np=1000)))
> loglik.true.est <- bl + log(mean(exp(loglik.true-bl)))
> loglik.true.est
[1] -22.18221
> loglik.true.se <- sd(exp(loglik.true-bl))/
+     exp(loglik.true.est-bl)
> loglik.true.se
[1] 0.1679821
```

Listing A.24: Processing the results from the MIF test for the example animal tracking problem. Showing how to pull the values of interest out of the `mif` object for the parameters estimated; the log-likelihood estimate and the standard error of the log-likelihood.

```
> theta
     d     h  drift   Xlon.0  Xlat.0  slon  slat   sOb
 1.003  1.000 0.050 -60.500  44.000 0.150 0.075 0.250
> theta.mif <- apply(sapply(track.Imif,coef),1,mean)
> theta.mif
          d            h         drift        Xlon.0
 0.99977323   1.00000000   0.05000000  -60.50000000
     Xlat.0         slon          slat           sOb
  44.00000000   0.14378084   0.07715031   0.25000000
> loglik.mif <- replicate(n=10,logLik
                         (pFilter(track.Imif[[1]],
```

```
+                          parms=theta.mif,Np=1000)))
> bl <- mean(loglik.mif)
> bl
[1] -21.71642
> loglik.mif.est <- bl + log(mean(exp(loglik.mif-bl)))
> loglik.mif.est
[1] -21.70551
> loglik.mif.se <- sd(exp(loglik.mif-bl))/
                       exp(loglik.mif.est-bl)
> loglik.mif.se
[1] 0.1513775
> theta.true <- coef((track.ex))
> theta.true
      d     h  drift   Xlon.0  Xlat.0  slon   slat    sOb
  1.000 1.000 0.050 -60.500  44.000 0.150  0.075  0.250
> loglik.true <- replicate(n=10,logLik(pFilter(track.ex,
                           parms=theta.true,Np=1000)))
> loglik.true.est <- bl + log(mean(exp(loglik.true-bl)))
> loglik.true.est
[1] -22.01209
> loglik.true.se <- sd(exp(loglik.true-bl))/
                       exp(loglik.true.est-bl)
> loglik.true.se
```

```
[1] 0.334381
```

Listing A.25: Processing the results from the IVP MIF test for the example animal tracking problem. Showing how to pull the values of interest out of the `mif` object for the parameters estimated; the log-likelihood estimate and the standard error of the log-likelihood.

```
    EID sealid    date          lat        lon      spd Timestep
X
1 35890 106709 14/06/2011 14:21 43.92794 -60.01687 1.9176288
0    0
2 35891 106709 14/06/2011 14:38 43.92359 -60.01379 2.1385149
1019 1019
3 35892 106709 14/06/2011 14:54 43.92142 -60.00716 2.1239551
982 2001
4 35893 106709 14/06/2011 15:10 43.92542 -60.00295 1.7354930
946 2947
5 35894 106709 14/06/2011 15:26 43.92339 -60.00785 3.3761050
939 3886
6 35896 106709 14/06/2011 15:59 43.92435 -59.98433 0.7476652
2010 5896
```

Listing A.26: The head of one of the sets of data supplied by OTN for one of the Atlantic Grey seals. The data has the date and time for the observation at the latitude and longitude positions, as well as the speed the animal was moving at. I've added Timestep and X to the data, where the Timestep is the seconds between observations and X is the cumulative time.

```
##state simulate function
```

```
real.track.state.sim <- function(x,t,parms,delta.t,...){
  ##get parameters
  gam=parms["gam"]
  if(parms["gam"]<0||parms["gam"]>1){
    if(parms["gam"]<0)
      parms["gam"] <- 0
    else
      parms["gam"] <- 1
  }
  if(parms["c"]<0||parms["c"]>1){
    if(parms["c"]<0)
      parms["c"] <- 0
    else
      parms["c"] <- 1
  }
  turn=rwrpcauchy(1, location=0, rho=parms["c"])
  ##setup Matrices
  ##define the turn matrix
  tx <- matrix(c(cos(turn),-sin(turn)
              ,-sin(turn),-cos(turn)),
              ncol=2,nrow=2)
  if(parms["slon"]<0)
    parms["slon"] <- 0
  if(parms["slat"]<0)
    parms["slat"] <- 0
```

```
  ##define the covariance matrix
  cv <- matrix(c(parms["slon"]^2,0,0,parms["slat"]^2),
               ncol=2,nrow=2)
  ##generate new states
  xnew <- rmvnorm(n=1,mean=(x[c("xlon","xlat")]+
                           (gam*(tx%*%(x[c("xlon","xlat")]
                           -x[c("xlon1","xlat1")])))),
                           sigma=cv)
  if(xnew[1]< -63.00)
    xnew[1] <- -63.00
  if(xnew[1]>-57.00)
    xnew[1] <- -57.00
  if(xnew[2]<40.00)
    xnew[2] <- 40.00
  if(xnew[2]>48.00)
    xnew[2] <- 48.00
  ##combine the new and state from 1 time step before together
  xnew <-c(xnew,unname(x[c("xlon","xlat")]))
  ##give the new states the correct names
  names(xnew) <- c("xlon","xlat","xlon1","xlat1")
  return(xnew)
}
```

Listing A.27: Observed data state(process) model, transitioning the state from time t to t+1.

```
##observation density calculator
```

```
real.track.obsev.den <- function(y,x,t,parms,log,...){
  ##get parameter
  nada <- parms["nada"]
  cv <- matrix(c(parms["nada"]^2,0,0,parms["nada"]^2),
               ncol=2,nrow=2)
  ##get density
  fnew <- dmvnorm(x=y[c("ylon","ylat")],
                  mean=x[c("xlon","xlat")],
                  sigma=cv,log=log)
  return(fnew)
}
```

Listing A.28: Observed data observation model, for calculating the likelihood of the state given the observed state.

Appendix B

Plots and Figures

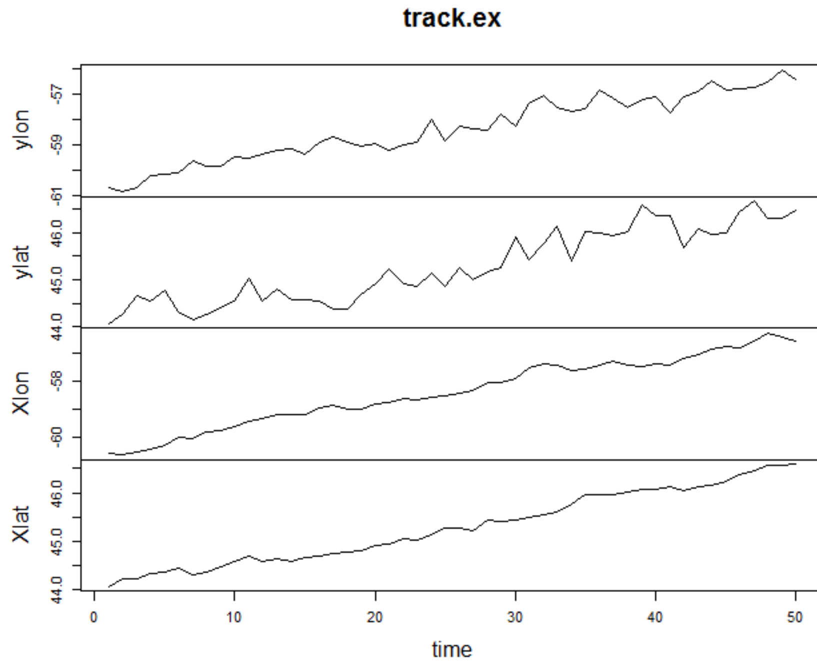


Figure B.1: The plot of the simulated data for the state and observations models, Xlon,Xlat state model, ylon,ylat observation model

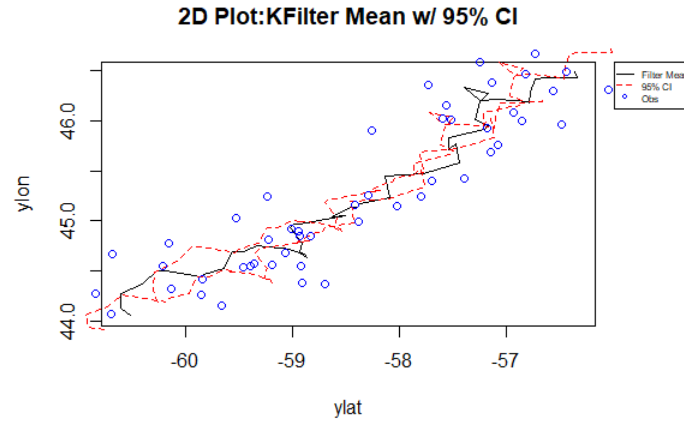


Figure B.2: The 2D plot of the Kalman Filter mean with 95% confidence intervals and observations

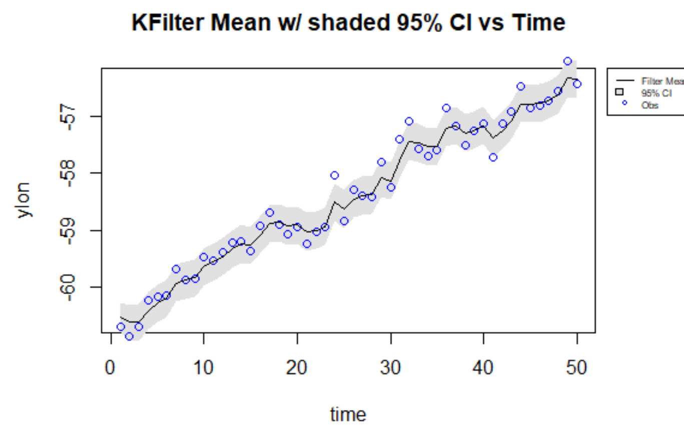


Figure B.3: The plot of the longitude variable Kalman Filter mean with shaded 95% confidence intervals and observations

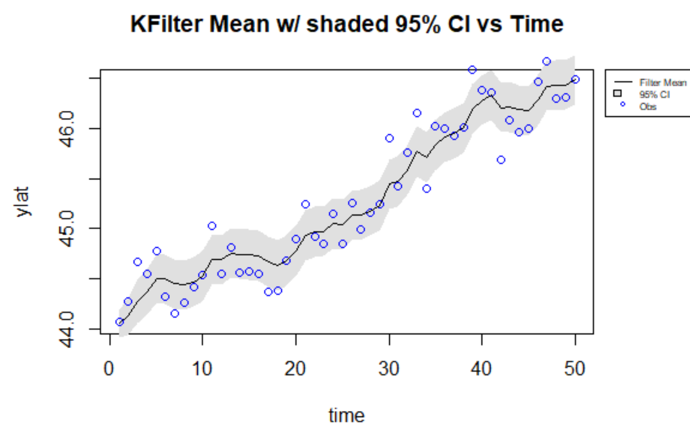


Figure B.4: The plot of the latitude variable Kalman Filter mean with 95% confidence intervals and observations

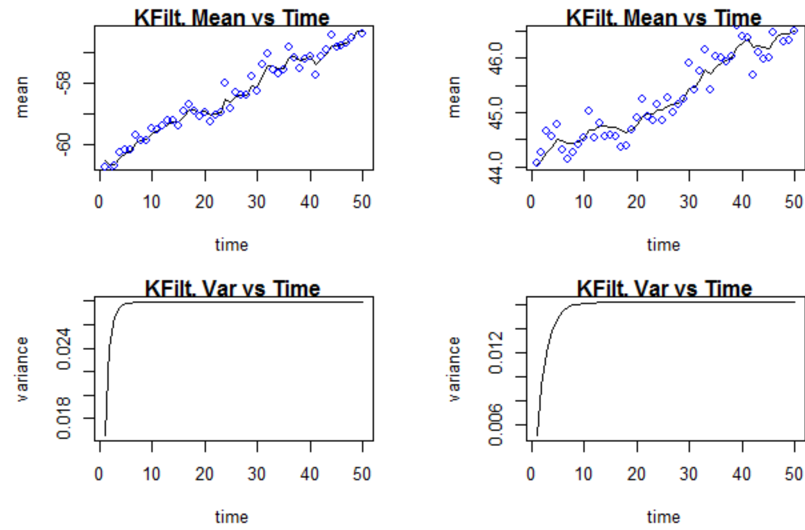


Figure B.5: The plots of the longitude (left) and latitude (right) variables Kalman filter mean (top) and variance (bottom) individual plots

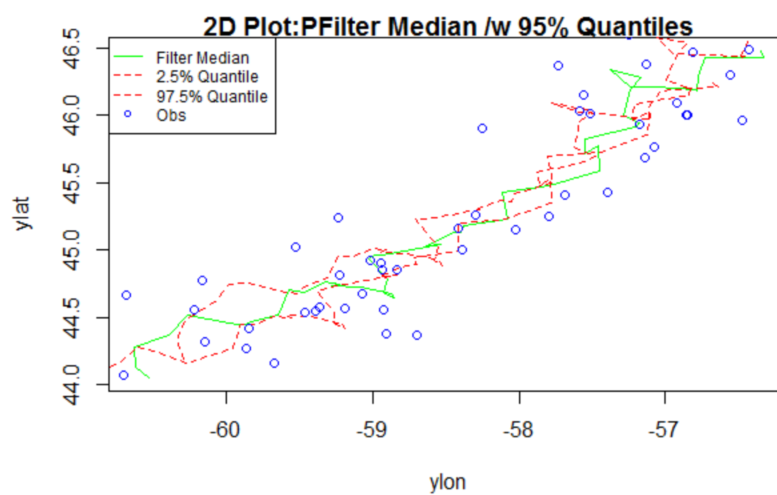


Figure B.6: The 2D plot of the particle filter median with 95% confidence intervals and observations

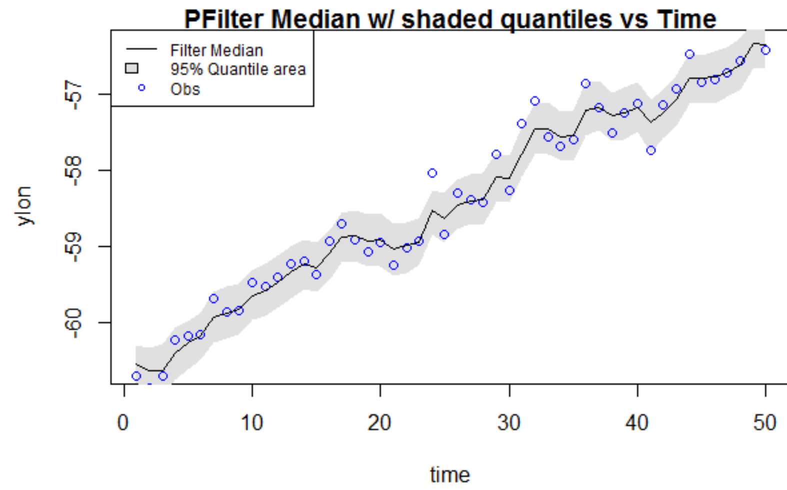


Figure B.7: The plot of the longitude variable particle filter median with shaded 95% confidence intervals and observations

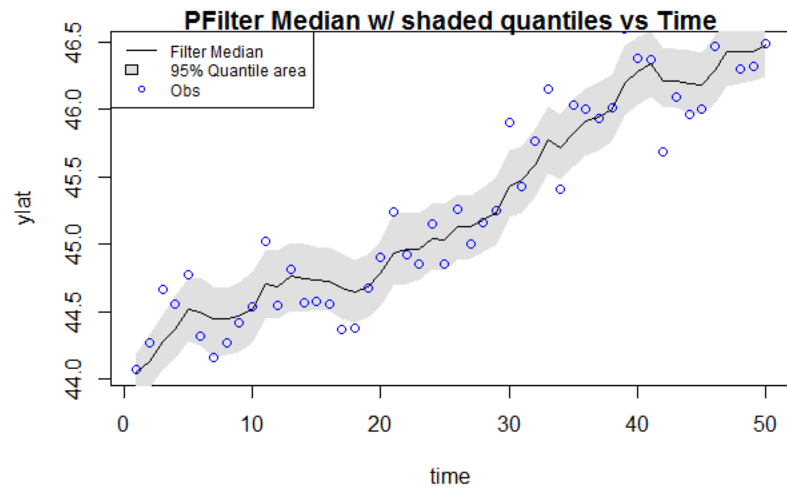


Figure B.8: The plot of the latitude variable particle filter median with shaded 95% confidence intervals and observations

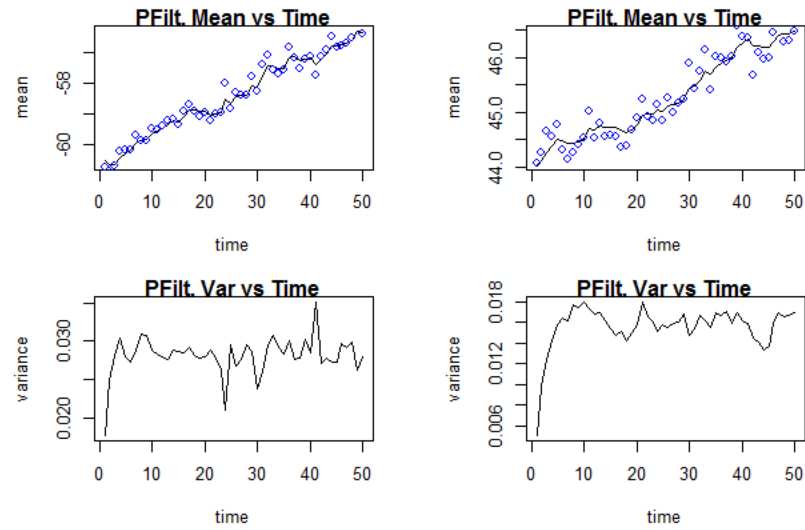


Figure B.9: The plots of the longitude (left) and latitude (right) variables particle filter mean (top) and variance (bottom) individual plots

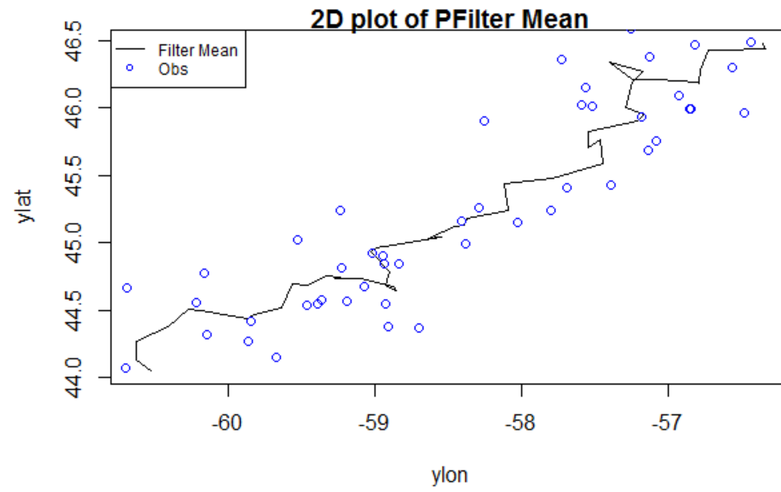


Figure B.10: The 2D plot of the particle filter mean and observations

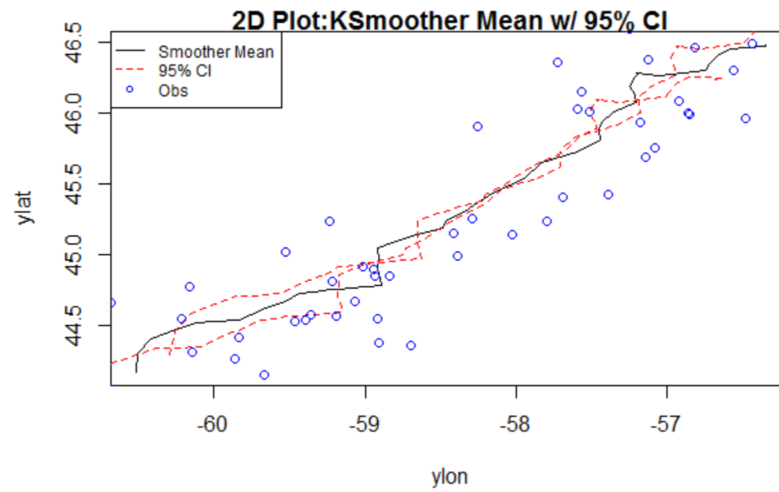


Figure B.11: The 2D plot of the results from the `kSmoother` function. The Kalman smoother mean (black) with 95% confidence intervals (red) are plotted with the recorded observations (blue).

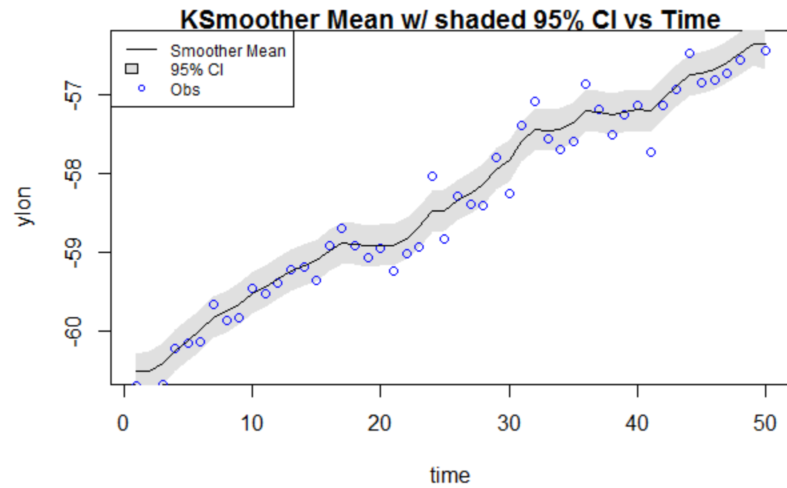


Figure B.12: The plot of the results from the `kSmoother` function for the longitude variable. The plot is of the Kalman smoother mean (black) with 95% confidence intervals (shaded) and the recorded observations (blue).

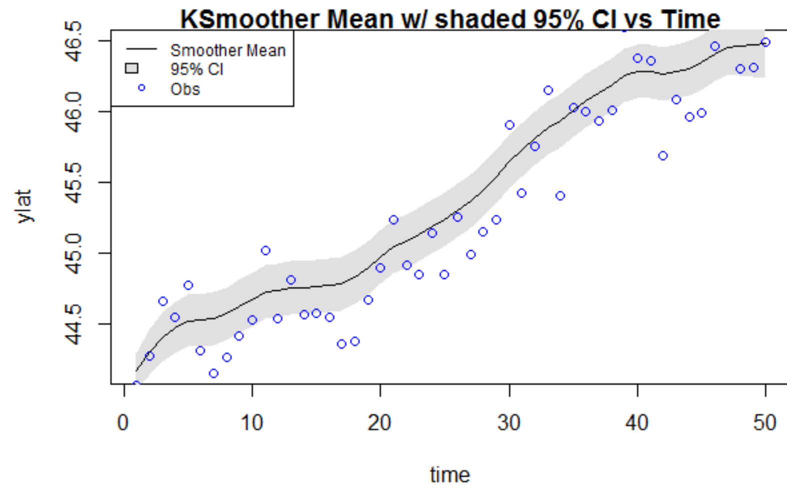


Figure B.13: The plot of the results from the `kSmoother` function for the latitude variable. The plot is of the Kalman smoother mean (black) with 95% confidence intervals (shaded) and the recorded observations (blue).

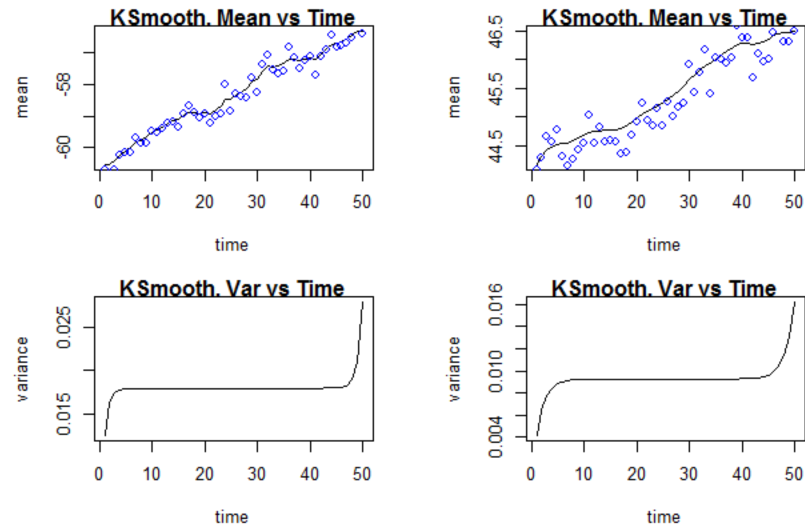


Figure B.14: The plots of the longitude (left) and latitude (right) variables Kalman smoother mean (top) and variance (bottom) individual plots from the `kSmoother` function results.

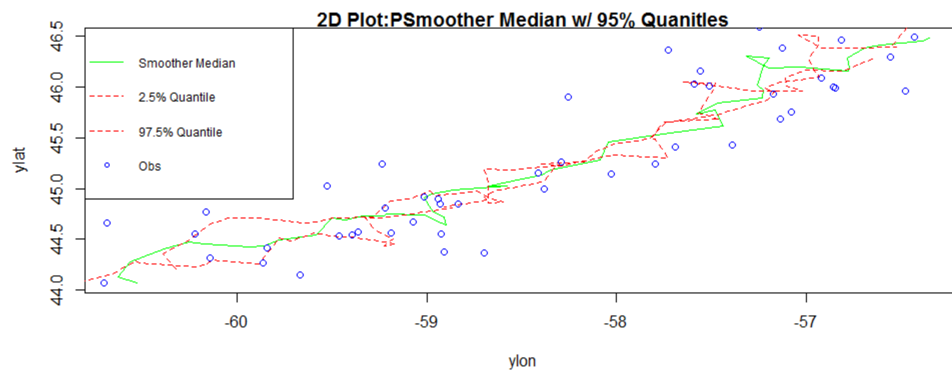


Figure B.15: The 2D plot of the Particle smoother median with 95% quantile area and observation points

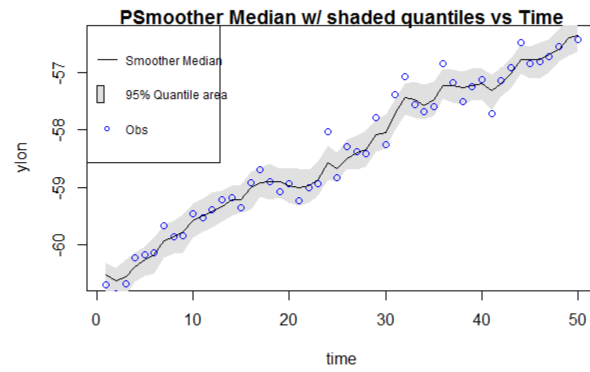


Figure B.16: The plot of the longitude variable Particle smoother median with 95% quantile area and observation points

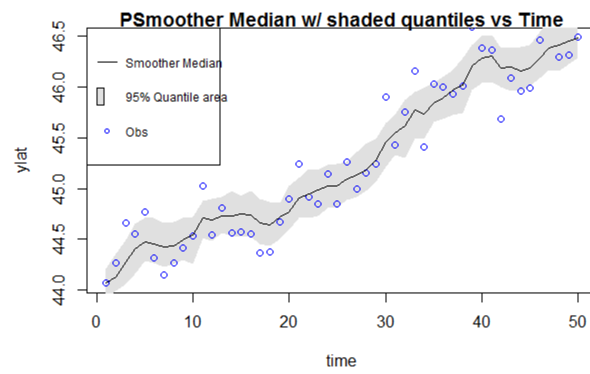


Figure B.17: The plot of the latitude variable Particle smoother median with 95% quantile area and observation points

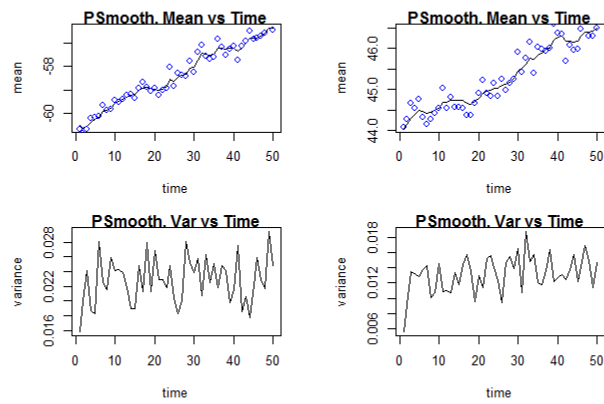


Figure B.18: The plot of the longitude (left) and latitude (right) variables Particle smoother mean (top) and variance (bottom) plotted against time.

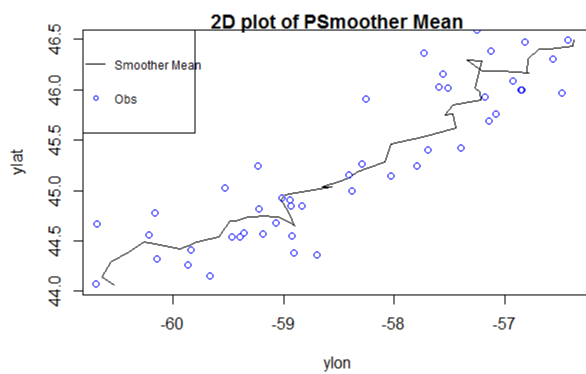


Figure B.19: The 2D plot of the longitude and latitude variables Particle smoother mean with the observation points

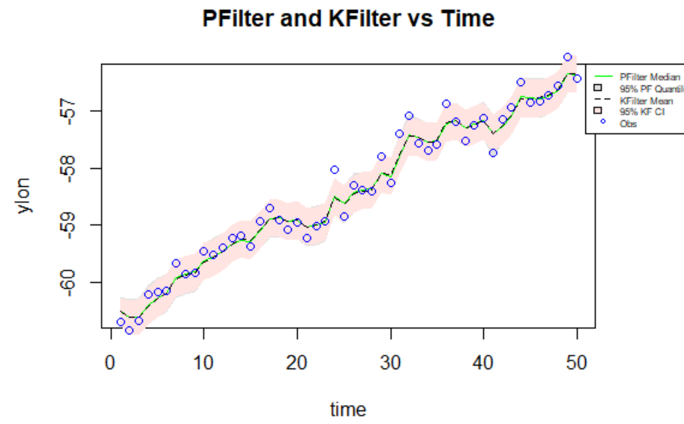


Figure B.20: The plot of the longitude variable Particle filter median vs. Kalman Filter mean

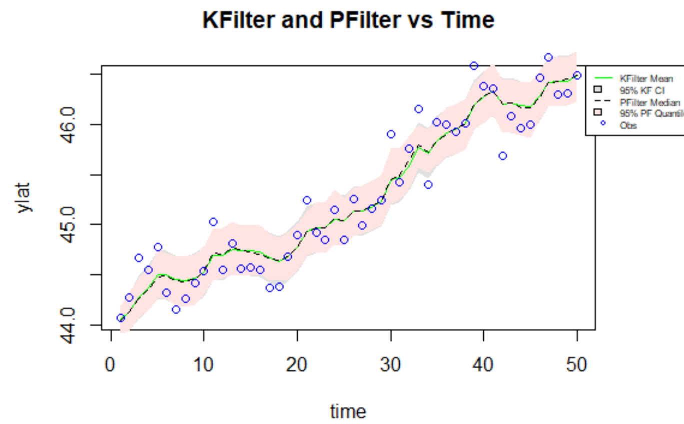


Figure B.21: The plot of the latitude variable Particle filter median vs. Kalman Filter mean

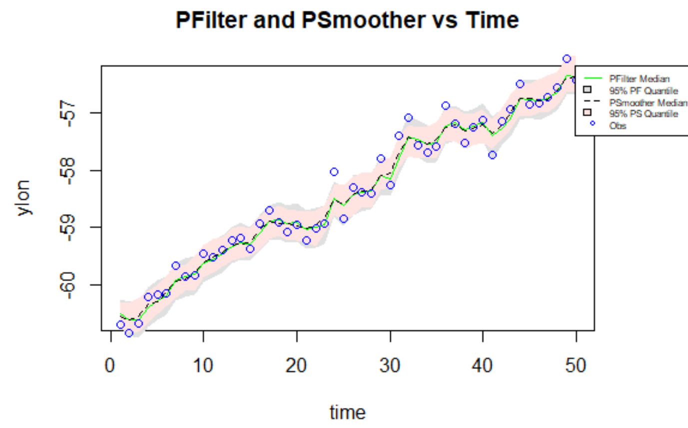


Figure B.22: The plot of the longitude variable Particle filter median vs. Particle Smoother median

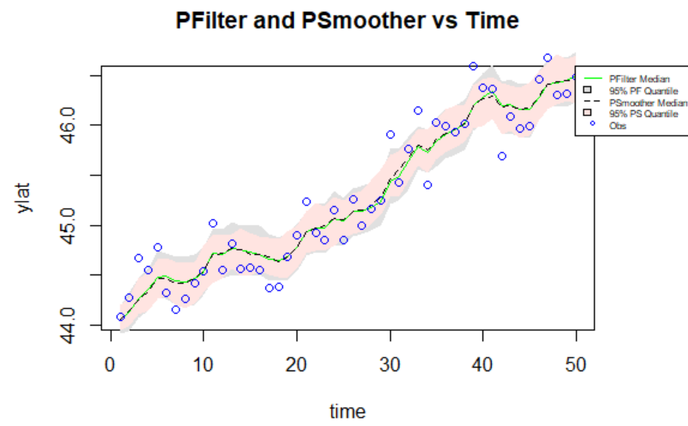


Figure B.23: The plot of the latitude variable Particle filter median vs.Particle Smoother Median

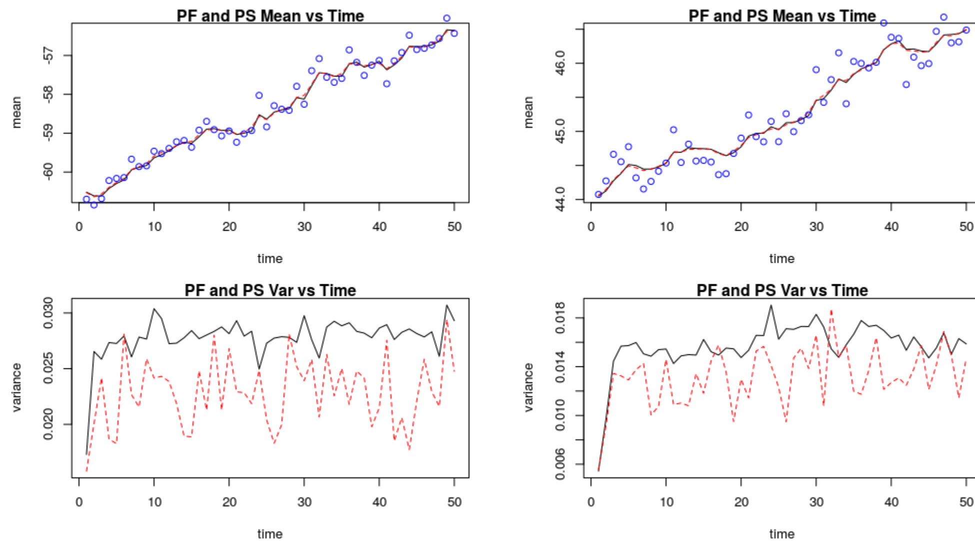


Figure B.24: The plots of the longitude (left) and latitude (right) individual variables Particle filter vs. Particle Smoother mean (top) and variance (bottom) plots each plotted against the time

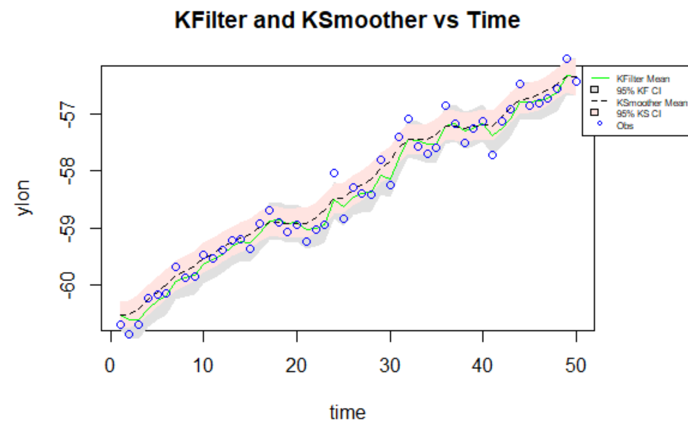


Figure B.25: The plot of the longitude variable Kalman filter mean vs. Kalman Smoother mean with the observations points

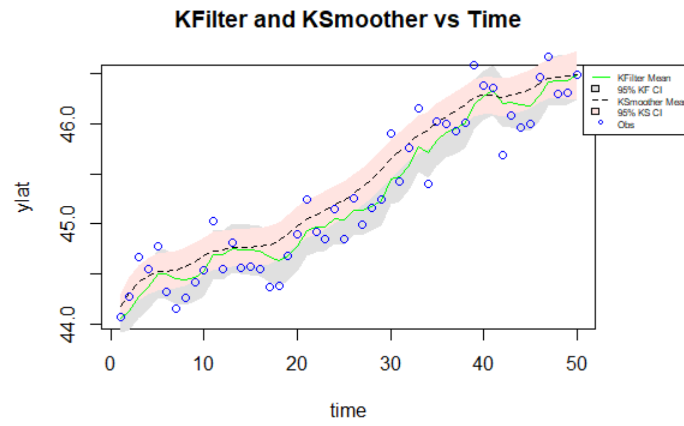


Figure B.26: The plot of the latitude variable Kalman filter mean vs. Kalman Smoother mean with the observations points

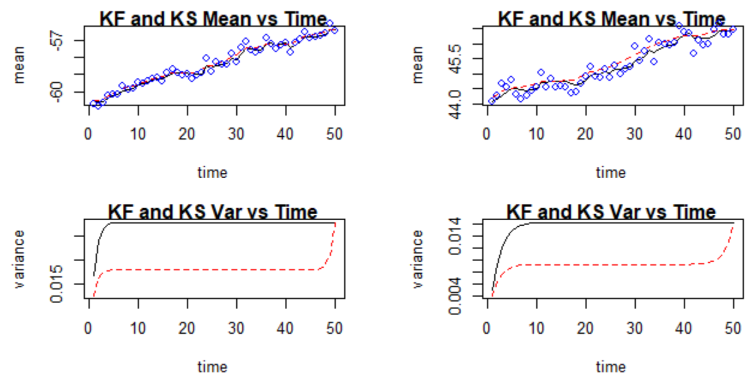


Figure B.27: The plot of the longitude (left) and latitude (right) variable Kalman filter mean vs. Kalman Smoother mean with the observations points (top) and Kalman filter variance vs. Kalman Smoother variance (bottom)

Appendix C

Comparisons Appendix

In this Appendix, the comparisons are presented for the results from Chapter 3. Using the example tracking problem to prove the equivalent of the procedures in the package `nLnG`, when the SSM being used is of the linear Gaussian class. We first show that the results from of the Kalman and Particle filter are equivalent to each other, then show that the Kalman filter and smoother have equivalent results and lastly show that the particle filter and smoother also have equivalent results as well.

The comparisons are all made by using the `compare.diag` method available in package `nLnG`. The `compare.diag` function takes two different `nLnG` procedure results (or they can be the same procedure, if you want to compare two Kalman filter results say) and produces comparison plots of the findings so that the user can easily view them together. The `compare.diag` function takes the following inputs, the two `nLnG` objects produced from the procedures, the number of variables in the data set and whether or not to produce a 2D plot of the variables. The comparison plots produced mirror the plots produced by the `filter.diag` and `smoother.diag` functions, except that the second set of results are overlaid onto the first set. Therefore a plot of each of the variable(s) with 95% CI against the time, a plot of the filter mean(s)

and variance(s) against the time and a 2D plot of the variables if the option has been selected. We can now look at the comparisons made for the example tracking problem used in Chapter 3. Let us start with the comparisons for the filter methods in package `nLnG`.

C.1 Filter Comparison

For the filter comparison, the results from the particle filter and Kalman filter results are compared to show the functions' equivalence under the linear Gaussian case.
Looking at the Figure C.1 closer you can see the particle filter median (green) and the Kalman filter mean (black) states match very well from the two procedures. With the particle filter median being practically directly on top of the Kalman filter mean in the plot. In Figure C.2 is comparing the two filter means and variances for each of the variables plotted against the time. In the bottom two variance plots you can see that the Kalman filter variance (red) crosses through the middle of the particle filter variance (black) for each the longitude and latitude position variables, this is what we hope to see when comparing the two procedure. The Kalman filter variance will always be much smoother and linear looking than the particle filter variance and overlaying them the Kalman filter variance could approximately be the mean of the particle filter variance. This is the expected result we were looking to show that when

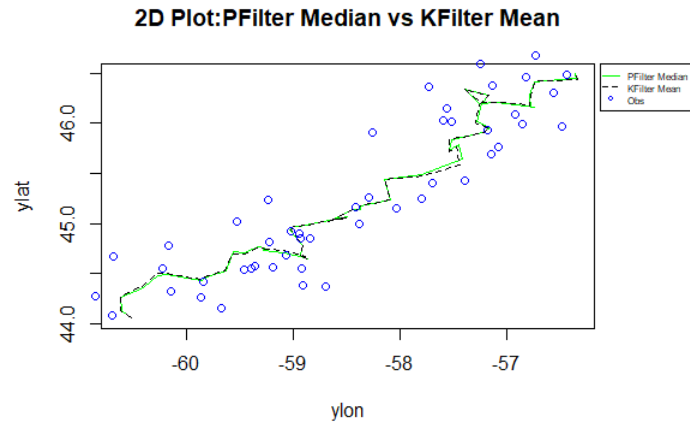


Figure C.1: The plot of the particle filter median, the Kalman filter mean and observations for the example tracking data, particle filter median (green), Kalman filter mean (black) and observation state (blue).

we have a linear Gaussian SSM the results are equivalent.

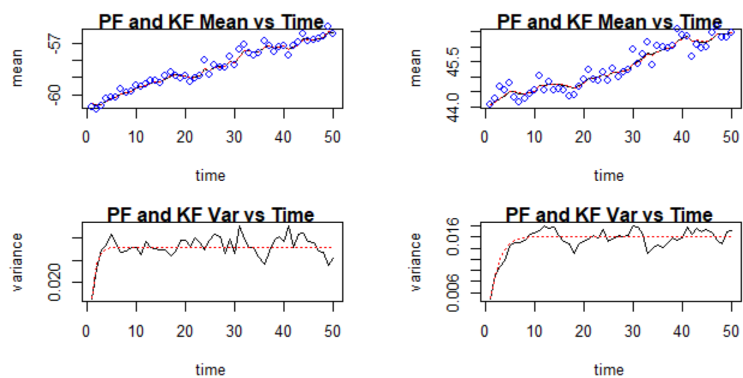


Figure C.2: The plots of the longitude and latitude variables Particle filter vs. Kalman filter mean and variance individual plots

C.2 Smoother Comparison

For the smoother comparison, the results from the particle smoother and Kalman smoother results are compared to show the functions equivalence under the linear Gaussian case. Looking at the Figure C.1 closer you can see the particle smoother median (green) and the Kalman smoother mean (black) states match very well from the two procedures. With the particle smoother median being very close to the

Kalman smoother mean throughout the plot. In Figure C.4 is comparing the two smoother means and variances for each of the variables plotted against the time.

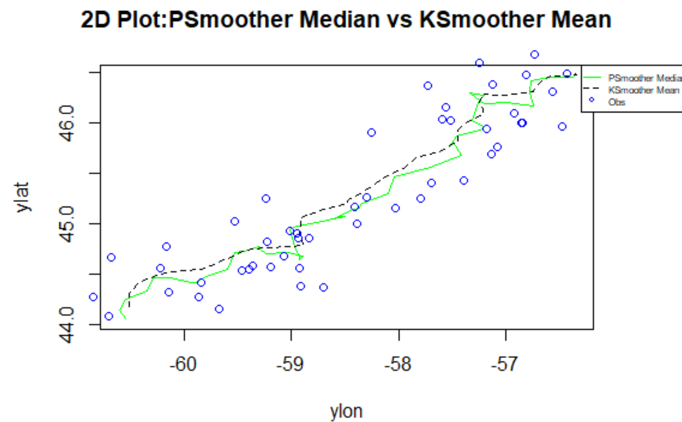


Figure C.3: The plot of the particle smoother median, the Kalman smoother mean and observations for the example tracking data, particle smoother median (green), Kalman smoother mean (black) and observation state (blue).

In the bottom two variance plots you can see that the Kalman smoother variance (red) crosses through the middle of the particle smoother variance (black) for each the longitude and latitude position variables, this is what we hope to see when comparing the two procedure. The Kalman smoother variance will always be much smoother and linear looking than the particle smoother variance and overlaying them the Kalman

smoother variance makes a floor of the lowest points for the particle smoother variance. Again, this is the expected result we were looking to show that when we have a linear Gaussian SSM that results are equivalent.

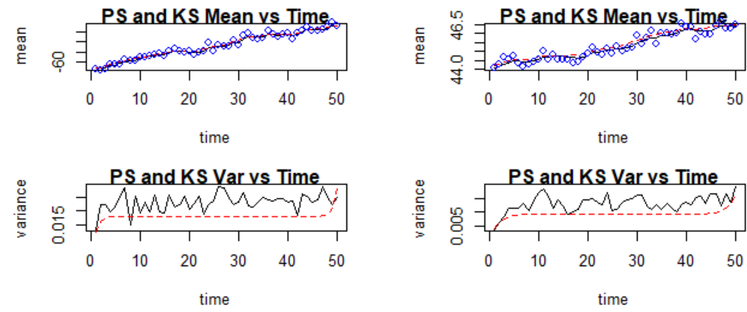


Figure C.4: The plots of the longitude and latitude variables Particle smoother vs. Kalman smoother mean and variance individual plots

List of Algorithms

1 The general filtering algorithm for State Space Models 15

2 The general forward-backward smoothing algorithm 22

3 `kFilter(object,parms,...)` 57

4 `pFilter(object,Np,parms,...)` 64

5 `kSmoother(kFilterd-nLnG object,parms,...)` 71

6 `pSmoother(object,parms,Nreal,Np,...)` 76

7 `stateAugementation(object,parms,Np,pars,pars.sd,mif.fail,...)` . . . 85

8 `mif(object,Nmif,Np,start,pars,ivps,cooling.factor,var.fact,...)` . . . 92

Bibliography

K. Andersen, A. Nielsen, U. Thygesen, H. Hinrichsen, and S. Neuenfeld. Using he particle filtering to geolocate atlantic cod (*Gadus morhua*) in the baltic sea, with special emphasis on determining uncertainty. *Canadian Journal of Fisheries and Aquatic Science*, 64 (4):618–627, 2007.

C. M. Bergman, J. A. Schaefer, and S. N. Luttich. Caribou movement as a correlated random walk. *Oecologia*, 123:364–374, 2000.

B. A. Block, H. Dewar, S.B. Blackwell, T.D. Williams, E.D. Price, C.J. Farewell, A. Boustany, S.L.H Teo, A. Seitz, A. Walli, and D. Fudge. Migratory movements, depth preferences, and thermal biology of altantic bluefin tuna. *Science*, 293:1310–1314, 2001.

G. Breed, D.P. Costa, I. Jonsen, P. Robinson, and J. Mills-Flemming. State-space methods for more completely capturing behavioral dynamics from animal tracks. *Ecological Modelling*, 235-236 (2012):49–58,, 2012.

G.A. Breed, I.D. Jonsen, R.A. Myers, and Leonard M.L. Bowen, W.D. Sex-specific, seasonal foraging tactics of adut grey seals (*Halichoerus grypus*) revealed by state-space analysis. *Ecology*, 90(11):3209–3221, 2009.

D. Costa, P. Robinson, J. Arnould, A. Harrison, S. Simmons, j. Hassrick, A. Hoskins, S. Kirkman, H. Oosthuizen, and et all. Villegas-Amtmann, S. Accuracy of argos locations of pinnipeds at-sea estimated using fastloc gps. *PLoS One 5*, 1:e8677, 2010a. URL `https://doi.org/10.1371/journal.pone.0008677`.

C. Dethlefsen, S. Lundbye-Christensen, and A. Christensen. *sspir: State Space Models in R*, 2012. URL `http://CRAN.R-project.org/package=sspir`. R package version 0.2.10.

A. Doucet and A. Johansen. A tutorial on particle filtering and smoothing: Fifteen years later. 2011.

A. Doucet, N.D Freitas, and N. Gordon. *Sequential Monte Carlo Methods in Practice*. Springer-Verlag, New York, 2001.

M. Dowd and R. Joy. Estimating behavioural parameters in animal movement models using a state augmented particle filter. *Ecology*, 92:395–408, 2011.

J. Durbin and S. Koopman. *Time Series Analysis by State Space Methods*. Oxford Univ Pr., 2001.

A.E. Gelfand and A.F.M Smith. Sampling-based approaches to calculating marginal densities. *Journal of the American Statistical Association*, 85:398–409, 1990.

J. Geweke. Bayesian inference in econmetrics models using monte carlo integration. *Econmetrica*, 57:1317–1339, 1989.

P. D. Gilbert. *Brief User's Guide: Dynamic Systems Estimation*, 2009. URL `http://cran.r-project.org/web/packages/dse/vignettes/dse-guide.pdf`.

S. J. Godsill, A. Doucet, and M. West. Monte carlo smoothing for nonlinear time series. *Journal of the American Statistical Assocation*, 99, No. 465, 2004.

N. J. Gordon, D. J. Salmond, and A. F. M. Smith. Novel approach to nonlinear/non-gaussian bayesian state estimation. *IEE-Proceedings-F*, 140:107–113, 1993.

J. Helske. *KFAS: Kalman Filter and Smoother for Exponential Family State Space Models.*, 2012. URL `http://CRAN.R-project.org/package=KFAS`. R package version 0.9.11.

E. Ionides, C. Bretó, and A. King. Inference for nonlinear dynamical systems. *Proceedings of the National Academy of Sciences of the United States of America*, 103(49), 18438, 2006.

E. Ionides, A. Bhadra, Y. Atchadé, and A. King. Iterated filtering. *The Annals of Statistics*, 39(3):1776–1802, 2011. doi: 10.1214/11-AOS886.

D.S. Johnson, J.M. London, M.A. Lea, and J.W. Durban. Continuous-time correlated random walk model for animal telemetry data. *Ecology*, 89(5):1208–1215, 2008.

I. Jonsen, R.A. Myers, and J. Mills-Flemming. Meta-analysis of animal movement using state-space models. *Ecology*, 84(11):3055–3063, 2003.

I. Jonsen, J. Mills Flemming, and R. Myers. Robust state-space modeling of animal movement data. *Ecology*, 86:2874–2880, 2005.

D. Kahle and H. Wickham. ggmap: Spatial visualization with ggplot2. *The R Journal*, 5(1):144–161, 2013. URL `https://journal.r-project.org/archive/2013-1/kahle-wickham.pdf`.

R. E. Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering*, 82(Series D):35–45, 1960.

P.M. Kareiva and N. Shigesada. Analyzing insect movement as a correlated random walk. *Oecologia*, 56:234–238, 1983.

A. King, E. Ionides, C. Bretó, S. Elner, B. Kendall, H. Wearing, M. Ferrari, M. Lavine, and D. Reuman. *pomp: Statistical inference for partially observed Markov processes (R package)*, 2010. URL `http://pomp.r-forge.r-rproject.org`.

G. Kitagawa. A self-organizing state-space model. *Journal of the American Statisitcal Association*, 93,443:1203–1215, 1998.

F. Leisch. Creating r packages: A tutorial. *Compstat 2008-Proceedings in Computational Statistics*, 2009.

J. Liu and M. West. Combining parameter and state estimation in simulation-based filtering. *Sequential Monte Carlo Methods in Practice (edited by A. Doucet, N. de Freitas, and N. J. Gordon)*, pages 197–224, 2001.

U. Lund and C. Agostinell. *CircStats: Circular Statistics, from "Topics in Circular Statistics" (2001)*, 2018. URL `https://CRAN.R-project.org/package=CircStats`. R package version 0.2-6.

I.M. Marsh and R.E. Jones. The form and consequences of random walk movement models. *Journal of Theoretical Biology*, 133:113–131, 1988.

A. Nielsen, K. Bigelow, M. Musyl, and J. Sibert. Improving light-based geolocation by including sea surface temperature. *Fisheries Oceangraphy*, 15(4):314–325, 2006.

A. Okubo and L.J. Gross. *Diffusion and ecological problems*, chapter Animal movements in home range, pages 238–266. Springer-Verlag, New York, New York, USA, 2002.

T. Patterson, L. Thomas, C. Wilcox, O. Ovaskainen, and J. Matthiopoulos. State-space models of individual animal movement. *Trends in Ecology & Evolution*, 23 (2):87–94, 2008.

G. Petris. An R package for dynamic linear models. *Journal of Statistical Software*, 36(12):1–16, 2010. URL `http://www.jstatsoft.org/v36/i12/`.

G. Petris and S. Petrone. State space models in r. *Journal of Statistical Software*, 41(4), 2011.

R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2014.

H. E. Rauch. Solutions to the linear smoothing problem. *IEEE Transactions on Automatic Control*, 8:371–372, 1963.

J. Roland, J.G. McKinnon, C. Backhouse, and P.D. Taylor. Even smaller radar tags on insects. *Nature*, 381:120, 1996.

F. Royer, J.M. Fromentin, and P. Gaspar. A state-space model to derivebluefin tuna movementand habitat from srchival tags. *Oikos*, 109 (3):473–484, 2005.

P. Turchin. *Quantitative Analysis of Movement: Measuring and Modeling Population Redistribution in Plants and Animals*. Sinauer Associates, 1998.

H. Wickham. *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, 2016. ISBN 978-3-319-24277-4. URL `https://ggplot2.tidyverse.org`.

J. Wong. Parameter estimation for nonlinear state space models, 2012.