Towards Efficient Estimations of Parameters in Stochastic Differential Equations

Rune Juhl

Kongens Lyngby 2011
IMM-M.Sc-2011-89

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

M.Sc. Thesis: ISBN 978-
ISSN 1601-233X

Summary

Stochastic differential equations are gaining popularity, but estimating the models can be rather time consuming. CTSM v2.3 is a graphical entry point which quickly becomes cumbersome. The present thesis successfully implements CTSM in the scriptable R language and exploit the independent function evaluations in the gradient.

Several non-linear model are tested to determine the performance running parallel. The best speed-up observed is 10x at a low cost of additional total CPU usage of a few percent.

The new CTSM interface lets a user diagnose erroneous estimations using the newly added traces of the Hessian, gradient and parameters. It lives within R and its very flexible environment where data preprocessing and post processing can be performed with the new CTSM.

Contents

CHAPTER 1

Background

This thesis was suppose to continue the analysis of clinical data in the DIACON project. Diabetes is a condition where the body is unable to regulate the glucose level in the blood. Glucose is the main source of energy for our cells, but the level must be sustained within limits to avoid irreversible organ damage such as diabetic retinopathy or neuropathy. Managing diabetes often relies on self-administration of insulin while monitoring the glucose level. The DIACON project aims at automating the infusion of insulin to sustain a stable level of glucose. The change of glucose level will follow some physical system but even assuming perfect measurements the measured levels will be subject to randomness. This systemic randomness is modelled by using stochastic differential equations (SDE). This was what I had undertaken, but I was challenged.

A major part in modelling is to identify and estimate parameters. For state space models where the system equations are SDEs the in-house grown Continuous Time Stochastic Modelling (CTSM) program by Niels Rode Kristensen and Henrik Madsen [14] is a way to formulate a model and estimate the parameters. The computational time depends both on the complexity of the model and the data and will quickly consume an hour or more. During model identification waiting for hours is not satisfactory. Realising the inherently serial optimisation relies on parallelisable calculations the speed of CTSM became my challenge.

Currently CTSM is presented to the user through a friendly graphical user interface designed in Java. This is an excellent entry point for some modellers but a cumbersome way to verify multiple models repeatedly while changing the underlying codebase. Thus I developed a simple interface in R.

The R interface was merely a tool for myself but a scriptable interface to

CTSM was highly requested. This was the birth of this thesis.

CHAPTER 2

CTSM

CTSM's main purpose is estimating parameters, but will also do simulation, smoothing, filtering and prediction. In the present thesis the parameter estimation is the focus as it is computationally heavy.

The real machinery of CTSM is a complex set of Fortran 77 routines. It depends on routines from the Harwell library, ODEPACK, LAPACK and BLAS.

2.1 A brief overview of the mathematics

The general non-linear model in state space form is

$$d\boldsymbol{X}_t = \boldsymbol{f}(\boldsymbol{X}_t, \boldsymbol{U}_t, t, \boldsymbol{\theta})dt + \boldsymbol{\sigma}(\boldsymbol{u}_t, t, \boldsymbol{\theta})d\mathbf{W}_t \tag{2.1}$$

$$\boldsymbol{y}_k = \boldsymbol{h}(\boldsymbol{x}_k, \boldsymbol{u}_k, t_k, \boldsymbol{\theta}) + \boldsymbol{e}_k \tag{2.2}$$

The state variable $\boldsymbol{X}_t$ is a n-dimensional real valued random variable, $\boldsymbol{u}_t$ m-dimensional real valued input, t a real valued time and $\boldsymbol{\theta}$ are the parameters. The vector function $\boldsymbol{f}(\cdot)$ is referred to as the drift and $\boldsymbol{\sigma}(\cdot)$ as the diffusion. The SDE is driven by a standard n-dimensional Wiener process (or a Brownian motion).

The Wiener process in eq. (2.1) has independent Gaussian distributed increments. CTSM assumes the conditional distribution of the k'th output is also Gaussian fully described by eqs. (2.3) and (2.4).

$$\hat{\boldsymbol{y}}_{k|k-1} = \mathbb{E}\left[\boldsymbol{y}_k | \mathcal{Y}_{k-1}, \boldsymbol{\theta}\right] \tag{2.3}$$

$$\boldsymbol{R}_{k|k-1} = \mathbb{V}\left[\boldsymbol{y}_k | \mathcal{Y}_{k-1}, \boldsymbol{\theta}\right] \tag{2.4}$$

Finding the optimal parameters is the non-linear optimisation problem.

$$\hat{\boldsymbol{\theta}} = \min_{\boldsymbol{\theta} \in \boldsymbol{\Theta}} \{-ln(L(\boldsymbol{\theta}; \mathcal{Y}_N | \boldsymbol{y}_0))\} \tag{2.5}$$

2.2 Optimising the objective function

The optimisation scheme used in CTSM is the well-known BFGS Quasi-Newton algorithm with an inexact line search. Specifically it is the VA13 routine from the Harwell library implemented back in 1975. It has been slightly modified but the calculations remain unchanged.

Quasi-Newton is a gradient based method where the Hessian is approximated. The Hessian is updated after every iteration using the BFGS updating scheme. Computationally this is done through two rank-1 updates. Not requiring a user defined Hessian is a major benefit as it is often impractical to determine. The required gradient is not even available analytically and thus approximated by forward or central finite difference. Forward difference is used during the first N iterations where the higher approximation error is less crucial. Moving closer to the solution the approximation is changed to the central finite difference approximation.

Evaluating the loss function once can be rather expensive. The forward approximation requires evaluating $loss(\boldsymbol{x}_{k-1} + \alpha \cdot \boldsymbol{p}_k)$ which is N evaluations of the loss function. The central approximation requires evaluating $loss(\boldsymbol{x}_{k-1} \pm \frac{\alpha}{2} \cdot \boldsymbol{p}_k)$ which is $2N$ evaluations. Clearly these are independent evaluations of a computationally expensive loss function which can benefit from running in parallel on a multi core processor.

Having an approximation of the Hessian and a finite difference approximation of the gradient the search direction is

$$\boldsymbol{p}_k = \boldsymbol{H}_k^{-1} \boldsymbol{g}_k. \tag{2.6}$$

The next point in the parameter space is

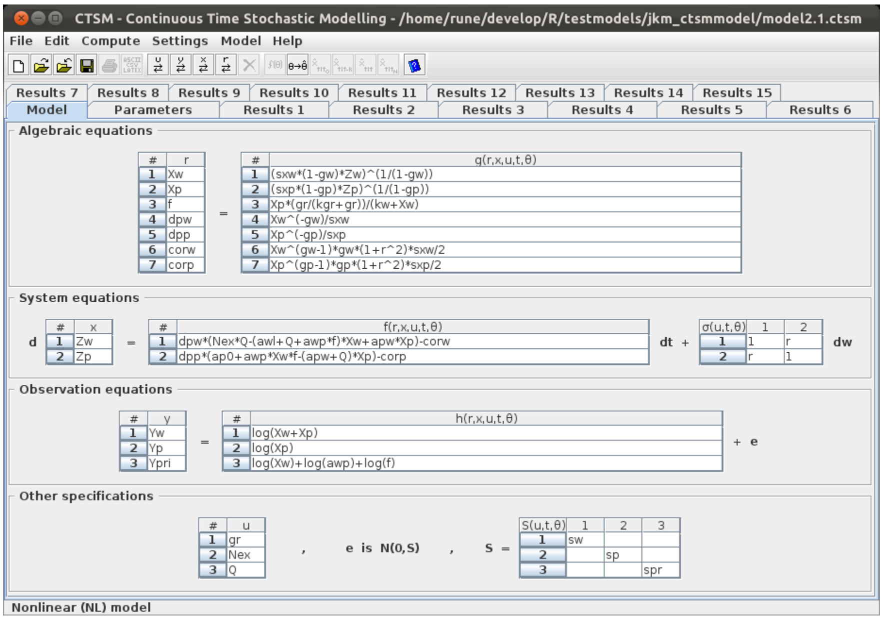
$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha \cdot \boldsymbol{p}_k \tag{2.7}$$

where α is a step length. With Quasi-Newton $\alpha = 1$ should always be tried first to ensure quadratic convergence. In general the optimal step length is that minimising the loss function along the search direction. This subproblem is solved inexactly and there are many such line search algorithms all trying to compute a step length such that the Armijo and Wolfe conditions are met[18]. This part is sequential.

The problems analysed with CTSM are likely to be small in dimension. Thus the matrix inversion and multiplications will not benefit from running parallel. Only the gradient will be calculated in parallel. The underlying code does not currently perform the calculations correctly in parallel.

2.3 The Graphical User Interface

Niels Rode Kristensen designed the current Java interface to CTSM. Figure 2.1 is the CTSM23 model specification of one the models used by [17]. The graphical user interface provides an easy way to specify a model and


CTSM - Continuous Time Stochastic Modelling - /home/rune/develop/R/testmodels/jkm_ctsmmodel/model2.1.ctsm
File Edit Compute Settings Model Help
Results 7 Results 8 Results 9 Results 10 Results 11 Results 12 Results 13 Results 14 Results 15
Model Parameters Results 1 Results 2 Results 3 Results 4 Results 5 Results 6

Algebraic equations

#	r
1	Xw
2	Xp
3	f
4	dpw
5	dpp
6	corw
7	corp

=

#	g(r,x,u,t,θ)
1	(sxw*(1-gw)*Zw)^(1/(1-gw))
2	(sxp*(1-gp)*Zp)^(1/(1-gp))
3	Xp*(gr/(kgr+gr))/(kw+Xw)
4	Xw^(-gw)/sxw
5	Xp^(-gp)/sxp
6	Xw^(gw-1)*gw*(1+r^2)*sxw/2
7	Xp^(gp-1)*gp*(1+r^2)*sxp/2

System equations

d

#	x
1	Zw
2	Zp

=

#	f(r,x,u,t,θ)
1	dpw*(Nex*Q-(awl+Q+awp*f)*Xw+apw*Xp)-corw
2	dpp*(ap0+awp*Xw*f-(apw+Q)*Xp)-corp

dt +

σ(u,t,θ)	1	2
1	1	r
2	r	1

dw

Observation equations

#	y
1	Yw
2	Yp
3	Ypri

=

#	h(r,x,u,t,θ)
1	log(Xw+Xp)
2	log(Xp)
3	log(Xw)+log(awp)+log(f)

+ e

Other specifications

#	u
1	gr
2	Nex
3	Q

, e is N(0,S) , S =

S(u,t,θ)	1	2	3
1	sw		
2		sp	
3			spr

Nonlinear (NL) model

Figure 2.1: Model specification

estimate it. This is very advantageous for students, external users and non power users in general. When used heavily as in fig. 2.1 one is quickly faced with having either many tabs or many saved model files. Changing the model is possible but it is not easy to recover a previously tried model. CTSM23 can only work on one model at a time and there is no possibility of queuing additional runs. The lack of scripting of batching is a major disadvantage when many models have to be analysed. The modeller must be present to start the next run.

2.3.1 Brief introduction to how it works

When the model has been specified as in fig. 2.1 it is symbolically analysed. Everything entered in CTSM23 are strings which are processed to verify the correctness of the mathematics and dependencies. To verify the mathematics every string is parsed through numerous tests e.g. counting parentheses and verifying the use of basic operators as =-*/.

The user must now specify initial values, bounds, estimation method and priors. CTSM is now ready to translate the model into valid Fortran 77 code which is saved in files in a working directory. For non-linear models the $\boldsymbol{f}(\cdot)$ and $\boldsymbol{h}(\cdot)$ are further differentiated with respect to the states and inputs. Technically this is automatic differentiation through source transformation done by the Jakef precompiler[9].

The Fortran 77 code is then compiled behind the scene and initiated by the CTSM Java interface. When completed the results are shown.

2.3.2 Some Problems with CTSM23

Being designed in Java the program is (to some extent) subject to the will of Oracle (previously Sun Microsystems). Some of the elements used have been deprecated and presently I am unable to alter any of the text boxes in the stock version and thus unable to specify any model.

Installation on 64 bit Linux based systems using the provided InstallAnywhere installation software is not possible due to a known bug in the installer.

The goal here is to provide a way to overcome a number of the current limitation.

2.4 R

R is a statistical language for analysing and visualising data [21]. It was conceived in 1993 in New Zealand by Ross Ihaka and Robert Gentleman. R is one of two modern implementations on the S programming language. R is an open source project under GNU which has gained much use in the academic world.

The R base is extended through thousands of packages developed by the community. The biggest source of packages is the Comprehensive R Archive Network (CRAN). The packages can use an underlying Fortran and C library.

Compared to the commercial MATLAB, R suffered in performance during loops. MATLAB has a Just in Time (JiT) compiler which greatly speeds up arithmetic loops. With the release of R version 2.14.0 a newly added byte compiler is now default. It is not automatically applied to user functions but a small test I conducted showed a 5 time speed up when applying the byte compiler on arithmetic functions.

Chapter 3

Pre-implementation considerations

Writing a serial program is quite easy. Going parallel forces the designer to think in parallel. CTSM relies on a number of libraries each of these libraries must be thread safe to be used in a threaded region.

3.1 Are the required libraries safe?

BLAS is the Basic Linear Algebra Subprograms which is the de facto standard for building blocks in numerical linear algebra. Especially the general matrix multiply routine `GEMM` is heavily used. BLAS exist as a reference implementation and in multiple optimised flavours for different architecture. Common for most is they are multi threaded and thus thread safe[3].

LAPACK is the Linear Algebra PACKage. LAPACK is built on top of BLAS and performs e.g. matrix factorisations. As of version 3.3 all routines in LAPACK are thread safe[16]. Unfortunately inspecting the source trunk of the recently released version 2.14.0 the included version of LAPACK is 3.1. All routines but `DLAMCH` (and three others not used by CTSM) are thread safe[15]. Although `DLAMCH` is present in the log-likelihood function it is never called during parameter estimation. Thus it is harmless here.

ODEPACK is a collection of nine solvers for initial value problems for ordinary differential equations (ODE)[10]. There is no information available on thread safety. The code is the original implementation from 1983 so it is likely it is not thread safe.

The Harwell library is only used for it legacy optimiser `VA13` and its dependencies. There will ever only be one instance within the program running.

3.2 OpenMP

There are a number of methods to produce parallel code. Two widely used are Message Passing Interface (MPI) and Open Multi-Processing (OpenMP). MPI is more difficult to implement but can work in a distributed memory setup - a cluster. OpenMP on the other hand is quite easy to implement but only on share memory systems, i.e. one multi core computer with a vast amount of memory. The size of the problems of interest and the size of the current servers at DTU there is no reason to use MPI over OpenMP.

The wonderful thing about OpenMP is one can gradually parallelise the code without major rewrites. One must remember that just because it is easy does not guarantee efficient parallel code. With OpenMP it is easy to get *false sharing*. Every core in a multi core CPU have its own small cache (L1) which is a piece of memory on the CPU between the core and the main memory. The data currently in the L1 cache is called a cache line. If two elements sit close on the same cache line and that cache line is loaded on multiple L1 caches then any update to one cache line will invalidate others. The other thread will be force to reload it from the memory. The solution is to pad those variables affected to get them on different cache lines. Congruency is likely not an issue here as only limited writing to shared arrays ever happen.

Since R 2.13.1 OpenMP is supported. Support is still eventually determined by the compiler, but R 2.13.1 accepts the `SHLIB_OPENMP` flag. Prior to version 2.13.1 small non portable hacks very required. These would raise warnings when building the package.

The GNU implementation of OpenMP is called GOMP. GOMP is working across platforms, but is broken for Microsoft Windows where the `threadprivate` clause is not working. The `SHLIB_OPENMP` flag is empty on Windows such that the code will never be compiled with OpenMP.

The required stack size may quickly be too little memory when using OpenMP. GNU OpenMP will allocate all local variables on the stack [7]. R has its own memory control and will terminate when the stack is almost fully used. In Linux systems the size of the stack can be changed by calling `ulimit -s unlimited` before starting R.

3.2.1 Directives

Implementing OpenMP is through directives (or pragmas). These pragmas are translated by a preprocessor before compiling the code. In fixed form

Fortran 77 all pragmas begin are stated in column 1 with `c$omp`. Thus if the code is compiled without OpenMP all pragmas are simply comments.

OpenMP offers many ways to control the level of parallelisation. Doing that requires knowledge of some pragmas. In Fortran 77 it is quick normal to use `COMMON` blocks of variables to avoid parsing too many variables through calls. A common block is shared between subroutines and act like a global variable. The `SAVE` attribute has a similar effect as it preserves the value of the variable between calls. Every `COMMON` and `SAVE` must be declared thread private using `c$OMP THREADPRIVATE(var1,var2,...)`. Upon entry to a parallel region each thread will have its own set of the thread private variables. The values are not copied and upon entry all the variables are uninitialised. This can be overcome by using the `copyin` clause which copies the values from the master thread to the corresponding variables in each thread.

Keeping common blocks private within threads is essential. The GNU implementation of OpenMP (GOMP) is broken on Microsoft Windows as the threadprivate clause is not working.

To specify a region in the code which should run in parallel is enclosed by Everything enclosed will be executed on each core. To run a loop in parallel

```
c$OMP PARALLEL
 ...
c$OMP END PARALLEL
```

Listing 1: OpenMP parallel clause

a `DO` clause is added. OpenMP will make sure the index variable is thread private.

3.2.2 Control the data sharing

Inside the parallel region OpenMP must know which variables are to be shared and which are to be private. There are two obvious clauses for this purpose: `SHARED(var1,var2,...)` and `PRIVATE(var3,var4,...)`. Shared variables have no restriction and can be altered by all threads. Care must be taken such that multiple thread are not updating the same variable. If so this can lead to a data race and corrupt the calculations.

`PRIVATE` variables gets a private instance in every thread. The variables are not initialised upon entry. This is accomplished by using the `FIRSTPRIVATE` clause. Variables declared firstprivate are initialised with the valued of the corresponding variables in the master thread just before entry.

3.3 Reference Classes

R has to class systems from the S language, S3 and S4. S3 is a simple class system. It is very easy and quite flexible to use. Far most of the packages for R are designed in S3 classes. S4 was introduced with the *methods* package. It is a much more rigorous and less flexible class than the S3 classes. It does provide more clear overview of the code as it is clear which methods acts on what.

R is written very functional programming, i.e. a function takes some inputs, process them and returns the result. Reference classes is an object oriented system with similarities to Java and C++. A class is an object with local variables (fields) and methods. Methods are acting on the object itself in contrary to the functional programming. I chose Reference Classes as I wanted a CTSM model where the users can add and remove equations, states etc. This is easily done when the methods modify the object itself. Also, it is possible to inherit classes much like S4 which will be used here.

One caveat is copying. Imaging having an instance of a model which should be copying to another variable. Normal R semantics would be `model2 <- model1`. This does not work with Reference classes. It will merely copy the reference to the underlying object. Modifying `model2` will thus show up in `model1`. A copy can be made, but must be done using the `copy()` method.

CHAPTER 4

Serial to Parallel

When using the OpenMP model only limited changes to the code are required to get it running.

There are two loops which can run in parallel. Only one run at the time, i.e. it depends on whether it is currently performing forward or central difference. The forward difference loop is shown in listing 2.

```
      DO 4 I=1,NX
         CALL FWDIFF(NX,X,I,XMIN,OD2,F,DF,INTS,NINT,DOUBLS,
     $        NDOUB,TMAT,NTMAT,IMAT,NIMAT,OMAT,NOMAT,NOBS,
     $        NSET,NMISST,EPSM,VINFO)
 4    CONTINUE
```

Listing 2: Wrapper for the forward difference approximation

CTSM stores the current parameter estimate, the input and t in and common block. Evaluating the loss function will change all of those. The common block must be declared as private in each thread. Previous attempts to implement OpenMP had already added the `threadprivate` clause to all common blocks in the CTSM code. However enabling OpenMP only caused CTSM to break down. Debugging parallel code changes the way the program is executed. Naturally one can only debug one thread at a time. The outcome of the code while debugging can be very different. In fact adding any kind of instrumentation to the code will interfere with its normal execution pattern.

I started debugging an OpenMP running parallel CTSM using Eclipse Photran which is a part of the Eclipse Parallel Tools Platform [5]. Having analysed a majority of the code I found out that CTSM would be returned by ODE integrator. CTSM would try to integrate multiple separate systems of ODEs but it turned into one major data race. The subroutines in ODEPACK relies heavily on common blocks shared within ODEPACK. These blocks of shared memory were shared in general over all running threads. The integrator would return as the system it was trying to solve was overwritten by another system - each thread fighting against each other. This was fixed by added the `threadprivate` clause to every common block and variable with the `SAVE` attribute.

At this point the code was running good. The included examples in the CTSM documentation were tried multiple times with consistent results - except for a few different results. The datasets for these two models are complete without missing observations.

Parallel CTSM was further tested using the model shown in fig. 2.1 on page 5 by Jan Kloppenborg Møller. The data supplied contains thousands of missing observations and the general model structure is vast compared to the examples from the documentation. CTSM returned prematurely. After much time spend on debugging the code another data race appeared. The log-likelihood function (loss function) counted the number of missing observations. This variable is a part of the arguments of the subroutine and can be traced back till listing 2 on the previous page where it is called `NMISST`. Due to the variable being shared all running threads were updating a single copy of it. Imagine two threads. One in the middle counting missing values and the other finishing. As the counting happens over multiple files the current count is added to the previous. The second thread finishes counting and updates `NMISST` to 5000 and continues. When the first thread will finish it will now update `NMISST` which is no longer 0 but 5000. CTSM failed as later checks showed too the data had too many missing observations.

The `FDF` subroutine calculated the function value and the gradient. Calculating the function value is the first call to the log-likelihood function, `LLIKE`, in listing 3 on the facing page. `NMISST` is update there and that number will not change to the solution was the remove the `NMISST` argument from `FWDIFF` in listing 2. Furthermore the gradient and vector info variables are now declared as shared in the final code in listing 3 on the next page.

The new model would now run. Much time was spend on going through the code to think about how each variable and argument are used in the parallel setting. After manually debugging and correcting two data races I learnt about thread analysers. Using both the Oracle Solaris Studio and Intel Thread Profiler the code was analysed for further data races and congruency. This process is very slow. Intel writes the execution time can be up to 300x normal speed. No further data races were found.

```
      XMIN = 1D-1
      CALL LLIKE(NX,X,INTS,NINT,DOUBLS,NDOUB,
     $            TMAT,NTMAT,IMAT,NIMAT,OMAT,NOMAT,NOBS,NSET,
     $            NMISST,FPEN,FPRIOR,EPSM,0,F,INFO)
      IF (INFO.NE.0) RETURN
      IF (MD.EQ.1) THEN
C
C     FORWARD DIFFERENCE APPROXIMATION TO GRADIENT .
C
C$OMP PARALLEL DO PRIVATE(I) FIRSTPRIVATE(X) SHARED(DF,VINFO)
      DO 4 I=1,NX
         CALL FWDIFF(NX,X,I,XMIN,OD2,F,DF,INTS,NINT,DOUBLS,NDOUB,
     $                TMAT,NTMAT,IMAT,NIMAT,OMAT,NOMAT,NOBS,NSET,
     $                EPSM,VINFO)
    4 CONTINUE
C$OMP END PARALLEL DO
```

Listing 3: Subset of the FDF subroutine calculating F and dF/dx

CHAPTER 5

The R Interface

CTSM in R (CtsmR here) has to major parts: (a) the user interaction part and (b) the part communicating with the Fortran codebase. This section will take out some of the important parts in the R code. Chapter 6 on page 21 goes through a number of models and the CtsmR implementations are shown there for reference.

5.1 User interface

CtsmR has 1 major class and 3 interface classes. The main class is called `ctsm.base` and is not exposed. The three interface classes are inheriting the `ctsm.base` class and provides model specifics. The three interfaces are: `ltictsm`, `ltvctsm` and `nlctsm` for the linear time invariant, linear time variant and non-linear models respectively. Specifics included in the interface classes are

- Interfaces for added equations and terms to the model
- Which dependences are allowed in the above matrices
- What goes in the internal A, B, C, D, SIGMAT and S matrices

The entire model specification will stay parsed by R and thus remain in the `language` data type. R's lists will be the internal data holder as lists are the object which can contain multiple `calls`. Matrices are unfolded (in column major as in Fortran) and stored in lists.

5.1.1 Adding an equation

A valid equation in CtsmR is a valid formula or expression in R. Writing something like `f<-a+b` will be evaluated at once so to keep that from happening one must `quote()` it. To avoid requiring the user the use `quote()` every time when adding an equation to the model the call is intercepted. The expressions can now also be added directly without first quoting them. Formulas like `f a+b` are simpler to cope with as they are not evaluated at once.

The `addequation` method in ctsm.base is finding all equations and inserted in the proper list. The left hand side becomes the name of the element and the right hand side the content. The expression `f==a+b` becomes `fvec[["X"]] = a+b` for the non-linear model. Adding equations/terms to the matrices is a bit more complicated. Currently the user must specify the position in the matrix.

There is no online check of illegal dependence in the equations.

5.1.2 Working on the equations

R is parsing the user entered equations before the CtsmR functions are actually called. Thus only mathematically valid expressions should appear. A parsed expression in R is essentially lists of lists as LISP. In fact `a+b` is behind the scenes `as.call(list(as.name("+"),as.name("a"),as.name("b")))`. CtsmR will have to run through the entire tree to any algebraic equations.

```r
codeWalker <- function(ex,node,...) {
   if (is.list(ex)) {
      for (j in 1:length(ex))
         ex[[j]] <- Recall(ex[[j]],node,...)
   }
   if (typeof(ex) == "language") {
      exx <- as.list(ex)
      for (i in 1:length(exx)) {
         ex[[i]] <- Recall(exx[[i]],node,...)
      }
   }
   # Reached a node
   ex <- node(ex,...)
   return (ex)
}
```

Listing 4: The code walker

Listing 4 is a general function to walk through the entire expression tree. It is used when end nodes must be handles as in substitution of equations.

Rather than walking through the expression tree it could be deparsed. Deparsing turns an expression into the corresponding string. All substitutions could then be done using regular expressions. Walking the trees seems more secure as the entire end note is compared to equation names. Regular expressions would need more protection which is indirectly given using the trees.

5.2 Processing to Fortran

There are two ways CtsmR will pass the model to the Fortran code. A general problem on the Windows platform is the required Fortran 77 are not available as standard. It it possible to get through the MinGW project but most people will not have this. Linux on the other hand may have it already, but if not it is quickly installed through a package manager.

To overcome the Windows issue two methods for evaluating the model is developed. The primary will work like CTSM23 and convert the problem into valid Fortran 77 which is compiled. The secondary method will work more like standard R, i.e. the pre-compiled Fortran code will through a C interface evaluate R functions or expressions within R.

Invoking the estimation starts a sequence to prepare the model for estimation. CtsmR determines the size of the model at this point. It is unlike CTSM23 never specified by the user. The algebraic equations are first checked for illegal dependence like implicit equations or cyclic dependence and then substituted into the model using the `codeWalker`. For non-linear models one extra step in done.

5.2.1 Differentiation

R can compute the symbolic derivatives of expressions. The automatic differentiation is thus no longer required. The R function to be used here is the simple `D()` which returns a simple call type. The non-linear case has two vector functions which are differentiated with respect to inputs and states. Listing 5 on the following page quickly differentiate a list of expressions in R The derivatives produced this way have been test both numerically and compared to symbolic differentiations in Maple with not mistakes.

The 4 matrices are now ready in the non-linear model. Finally ctsm.base is called to perform the steps common to all models types.

5.2.2 Compiling the model

The default is to write the model out in Fortran code and compile it. The user defined variables names must first be converted into the internal vector notation. Having lists of state names, parameter names and input names the `codeWalker()` is invoked to process the tree with a special leaf node

```
diffVectorFun <- function(f,x) {
   nf <- length(f)
   nx <- length(x)
   # Output as a list
   jac <- vector("list",nf*nx)

   # Column major
   k <- 0
   for (j in 1:nx)
      for (i in 1:nf)
         jac[[k<-k+1]] <- D(f[[i]],x[j])
   jac
}
```

Listing 5: R code to differentiate a list of expressions

function. Only numbers and variables are converted not intrinsic functions like *sin*, *exp* etc. where the generic function in Fortran are used.

The lists containing the Jacobians are deparsed to strings. A line can quickly become longer than the allowed 72 columns in fixed-form Fortran. Ever line is spilt into chunks of 64 characters. If splits occurred the subsequent lines are written with a continuation mark.

Finally the model is compiled in a temporary directory. If successful, a reference to the library file is saved.

5.2.3 The Windows alternative

This alternative was intended to become the primary link between the model and Fortran. However R is very single threaded. This is a problems as CTSM will have to evaluate the model multiple times in parallel to take advantage on the speed-up. The Fortran code might run in parallel but all requests to R will be queued thus reverting everything back to serial.

Another problem is R being an interpreted language. Unlike compiled Fortran R contains multiple layers which are involved in the calculations. When using R version 2.14.0 the byte compiler will be used and gain the additional speed-up. My tests have shown Fortran is much faster, but byte compiled R code is some 5 times faster. First the calls are converted into a function using the `calltoFunction()` in ctsm.base. This is only necessary because the byte compiler only handles functions.

R calls a C interface, which calls the main Fortran code. Every time the matrices are evaluated the Fortran code calls a Fortran wrapper, which calls the C wrapper which evaluates the expressions in the R environment.

5.3 Exposed classes

Most of the code is entirely internal. The three model classes are available. A new instance of the class is done by `model <- [ltv,lti,nl]ctsm$new()`.

Having a model the following is possible

- `$[add,remove]drift` - Add drift term(s)
- `$[add,remove]diffusion` - Add diffusion term(s)
- `$[add,remove]measure` - Add measurement equation(s)
- `$[add,remove]noise` - Add noise terms(s)
- `$[set,get]options` - Set options
- `$gentemplate` - Generate a template for entering initial values
- `$[set,get]configpars` - Set the parameter configuration
- `$estimate` - Estimate
- `$simulate` - Simulate
- `$smooth` - Smooth
- `$predict` - Predict

5.4 Diagnostics

Diagnosing an optimisation in CTSM23 is next to impossible. The user has no other option but to look at CTSM23 while running and writing the parameter trace down by hand.

CtsmR is tracing the optimisation. Currently the following are saved per iterations:

- The diagonal elements in the approximated Hessian
- The finite differences approximation of the gradient
- The scalar step length
- The current parameters
- The function value

Those five information can provide valuable insight to the optimisation. It should be noted that the Hessian, gradient and parameters are given in the optimisation space and not the original. The values can easily be back transformed.

CHAPTER 6

Experiments

5 non-linear models were estimated multiple times to verify the correctness of the new CTSM running in parallel. The number of threads used is varied from 1 to 20. The smaller models were typically estimated multiple times at for each number of threads. The larger models consume too much time and all tests cannot be completed within the 24 limit at the G-bar. The G-bar was chosen because there are 12 servers each with two 12 cores available and the load is in general very low.

The time spend for the estimation was saved to study whether running in parallel actually is faster. The timing is done in R using the `system.time()` function which returns 3 (+ 2 hidden) time measurements. The first is the CPU time required by the estimation, i.e. the total CPU time. The third is the elapsed time on the wall clock, call it T_p where p is the number of cores used. T_1 is the time used for the estimation using a single core. The speedup in parallel is then

$$S_p = \frac{T_1}{T_p}. \tag{6.1}$$

The overall load on the servers where manually checked every now and again to ensure the timing would not be corrupted do to other users' usage. The timing is very consistent when the requested cores are not performing other tasks. The consistency is expected as the estimation is a fully deterministic process.

6.1 Non-linear model of a fed-batch bioreactor

This model is included in the original CTSM and is described in the user guide parenciterode2003.

The biomass concentration X, substrate concentration S and volume V in a fed-batch bioreactor is modelled and formulated as an SDE in state space. Equations (6.2) and (6.3) are the system and observation equations respectively.

$$d\begin{pmatrix} X \\ S \\ V \end{pmatrix} = \begin{pmatrix} \mu(S)X - \frac{FX}{V} \\ -\frac{\mu(S)X}{Y} + \frac{F(S_F - S)}{V} \\ F \end{pmatrix} dt + \begin{bmatrix} \sigma_{11} & 0 & 0 \\ 0 & \sigma_{22} & 0 \\ 0 & 0 & \sigma_{33} \end{bmatrix} d\boldsymbol{\omega}_t \tag{6.2}$$

$$\begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}_k = \begin{pmatrix} X \\ S \\ V \end{pmatrix}_k + \boldsymbol{e}_k, \quad \boldsymbol{e}_k \in N(\boldsymbol{0}, \boldsymbol{S}), \quad \boldsymbol{S} = \begin{bmatrix} S_{11} & 0 & 0 \\ 0 & S_{22} & 0 \\ 0 & 0 & S_{33} \end{bmatrix} \tag{6.3}$$

F is an input and $\mu(S)$ is a growth rate and is given

$$\mu(S) = \mu_{max} \frac{S}{K_2 S^2 + S + K_1}. \tag{6.4}$$

The rest are parameters - some of which will be estimated.

The implementation in CtsmR is shown in listing 6. At this point CtsmR

```
# Create a NL model
model <- nlctsm$new()
# Add the growth equation
model$addequation(mu==mumax*S/(K2*S^2+S+K1))
# Add a state equation
model$addstate(X==mu*X-F*X/V)
# Add two states equations at once
model$addstate(S==-mu*X/Y+F*(SF-S)/V,V~F)
# Add diffusion terms
model$adddiffusion(1,sig11,5,sig22,9,sig33)
# Add the measurement equations
model$addmeas(y1~X,y2~S,y2~V)
# And the noise terms
model$addnoise(1,s11,5,s22,9,s33)
```

Listing 6: CtsmR implementation of a fed-batch reactor model

can be asked to generate a template for the specification of initial values, bounds and if it should be estimated. Invoke `model$gentemplate()` to get the template in listing 7 on the next page. The initial values and their bounds are taken from table B.1 [13, p. 50] and the original CTSM model file and shown in table 6.1 on the facing page. The lower and upper bound for K_2, S and Y are ignored if supplied. 11 parameters are to be estimated and 3 are fixed to a value.

Two options are changed from the default values. The data is loaded and the model is estimated in R as shown in listing 8 on the next page.

```
[MODEL]$configpars( #
#    States     ,       Method    ,   Lower    ,   Inital    ,   Upper    #
      "X0"     ,c(        1       ,     0      ,     0       ,     0      ),
      "S0"     ,c(        1       ,     0      ,     0       ,     0      ),
      "V0"     ,c(        1       ,     0      ,     0       ,     0      ),
#   Parameter ,       Method    ,   Lower    ,   Inital    ,   Upper    #
      "K1"     ,c(        1       ,     0      ,     0       ,     0      ),
      "K2"     ,c(        1       ,     0      ,     0       ,     0      ),
      "SF"     ,c(        1       ,     0      ,     0       ,     0      ),
      "Y"      ,c(        1       ,     0      ,     0       ,     0      ),
      "mumax"  ,c(        1       ,     0      ,     0       ,     0      ),
      "sig11"  ,c(        1       ,     0      ,     0       ,     0      ),
      "sig22"  ,c(        1       ,     0      ,     0       ,     0      ),
      "sig33"  ,c(        1       ,     0      ,     0       ,     0      ),
      "s11"    ,c(        1       ,     0      ,     0       ,     0      ),
      "s22"    ,c(        1       ,     0      ,     0       ,     0      ),
      "s33"    ,c(        1       ,     0      ,     0       ,     0      )
)
```

Listing 7: Templete for configuring the parameters

	Method	Lower	Initial	Upper
X0	1	0	1	2
S0	1	0	0.25	1
V0	1	0	1	2
K1	1	0	0.03	1
K2	0		0.5	
SF	0		10	
Y	0		0.5	
mumax	1	0	1	2
sig11	1	0	0.01	1
sig22	1	0	0.01	1
sig33	1	0	0.01	1
s11	1	0	0.1	1
s22	1	0	0.1	1
s33	1	0	0.1	1

Table 6.1: Parameter configuration for listing 6

```
# Change the ODE solver to Adams and the number of iterations in EKF to 1
model$setoptions(con=list(solutionMethod="adams",nIEKF=1))
# Load the data
data <- read.csv("nlex/sde0_1.csv", header=FALSE, sep=";")
# Slight reformat
data <- list(time=data[,1],inputs=data[,-1])
# Estimate
res <- model$estimate(data, sampletime=0, interpolation=0, threads=11)
```

Listing 8: Load data and estimate the model

This model is rather small and thus all computations are repeated 10 times. The estimation is done using from 1 to 20 threads. All estimations were compared to the very first estimation. The snippet in listing 9 is shown here but is used for all models.

```
rep <- 10
thr <- 1:20
# Prepare lists for the results
res <- vector("list", rep*length(thr))
times <- vector("list", rep*length(thr))
# Loop away
for (th in thr)
   for (j in 1:rep) {
      times[[(th-1)*rep+j]] <- system.time(res[[(th-1)*rep+j]]
          <- model$estimate(data, sampletime=0, interpolation=0, threads=th))
   }
```

Listing 9: Repeated estimation and timing

Figure 6.1 shows the wall time as a box plot. Clearly the timing is very consistent. The parameter subspace has 11 dimensions corresponding to the 11 free parameters. Thus the gradient requires independent point calculations in 11 directions. As expected the fastest estimation time is achieved when using 11 cores. The speed-up is almost 7x.

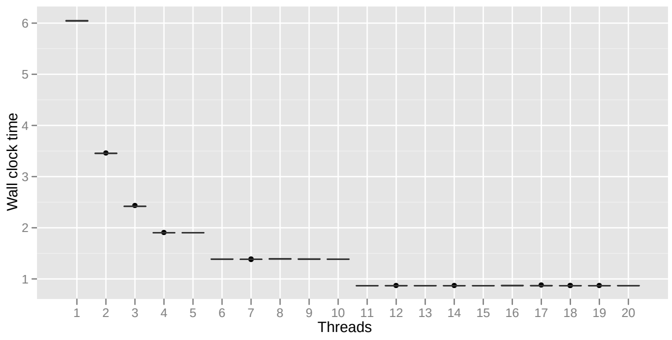


Figure 6.1: Wall time during estimating the fed-batch bioreacter model

Figure 6.2 on the facing page is the total CPU time used while estimating the model. This is also a view on the efficiency. It is not free to use more cores as the total CPU time is increasing with increasing number of cores. The large dip at 11 cores is expected as all cores will be used during the calculation of the gradient. The total CPU time is increased by 54%. In absolute numbers it is only about 3 seconds increase. The extra time is overhead going in and out of a parallel region. The simplicity of the model and size of data causes the overhead to be significant.

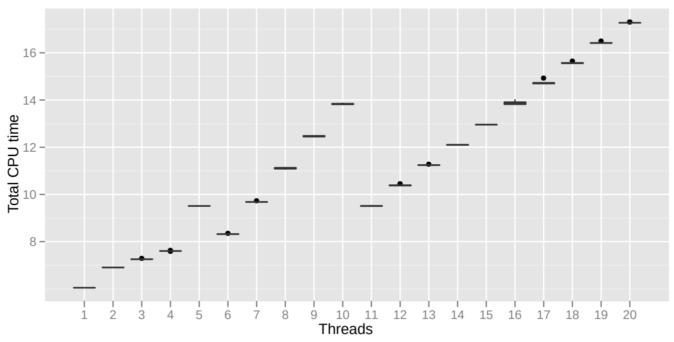


Figure 6.2: Total CPU time during estimation of the fed-batch bioreactor model

All 200 estimations gave identical results. The results from CtsmR are compared to CTSM in tables 6.2 and 6.3. The values are very close.

	CTSM	CtsmR
Objective function	-388.4856754136	-388.4856757680
Iterations	48	47
Function evaluations	74	63

Table 6.2: Optimisation Results

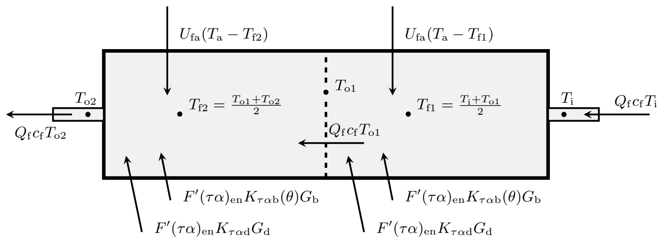
	CTSM		CtsmR	
Name	Estimate	Std. dev.	Estimate	Std. dev.
X0	1.009 55	$1.059\,68 \times 10^{-2}$	1.009 55	$1.507\,96 \times 10^{-2}$
S0	$2.383\,47 \times 10^{-1}$	$9.308\,00 \times 10^{-3}$	$2.383\,47 \times 10^{-1}$	$9.366\,96 \times 10^{-3}$
V0	1.003 95	$7.917\,15 \times 10^{-3}$	1.003 95	$9.370\,32 \times 10^{-3}$
mumax	1.002 24	$2.843\,86 \times 10^{-3}$	1.002 24	$4.177\,70 \times 10^{-3}$
K2	5×10^{-1}		5×10^{-1}	
K1	$3.162\,94 \times 10^{-2}$	$1.313\,94 \times 10^{-3}$	$3.162\,94 \times 10^{-2}$	$2.189\,35 \times 10^{-3}$
Y	5×10^{-1}		5×10^{-1}	
SF	$1.000\,00 \times 10^{1}$		$1.000\,00 \times 10^{1}$	
sig11	$1.552\,98 \times 10^{-27}$	$7.553\,69 \times 10^{-26}$	$1.563\,00 \times 10^{-26}$	$7.213\,50 \times 10^{-25}$
sig22	$1.765\,42 \times 10^{-6}$	$1.330\,94 \times 10^{-5}$	$8.013\,09 \times 10^{-7}$	$6.973\,23 \times 10^{-6}$
sig33	$1.149\,93 \times 10^{-8}$	$1.268\,31 \times 10^{-7}$	$1.788\,68 \times 10^{-8}$	$1.893\,28 \times 10^{-7}$
s11	$7.524\,75 \times 10^{-3}$	$9.997\,02 \times 10^{-4}$	$7.524\,79 \times 10^{-3}$	$1.089\,41 \times 10^{-3}$
s22	$1.063\,61 \times 10^{-3}$	$1.383\,74 \times 10^{-4}$	$1.063\,61 \times 10^{-3}$	$1.410\,16 \times 10^{-4}$
s33	$1.138\,85 \times 10^{-2}$	$1.530\,64 \times 10^{-3}$	$1.138\,85 \times 10^{-2}$	$1.531\,12 \times 10^{-3}$

Table 6.3: Estimation Results

6.2 Heat dynamics of solar collectors

This test model is kindly provided by Peder Bacher [2].

The problem here is estimating the parameters in a non-linear model of a solar collector. At first the collector is seen as a single compartment where the temperature is modelled as the average temperature of the in and outflow assuming constant temperature of the inflow. The one compartment model is expanded to n_c compartments. The two compartment model is shown in fig. 6.3. The n_c compartment model is given in eq. (6.5). These are the state



$U_{\mathrm{fa}}(T_{\mathrm{a}}-T_{\mathrm{f2}})$
$U_{\mathrm{fa}}(T_{\mathrm{a}}-T_{\mathrm{f1}})$
T_{o1}
T_{o2}
$T_{\mathrm{f2}}=\frac{T_{\mathrm{o1}}+T_{\mathrm{o2}}}{2}$
$T_{\mathrm{f1}}=\frac{T_{\mathrm{i}}+T_{\mathrm{o1}}}{2}$
T_{i}
$Q_{\mathrm{f}}c_{\mathrm{f}}T_{\mathrm{i}}$
$Q_{\mathrm{f}}c_{\mathrm{f}}T_{\mathrm{o2}}$
$Q_{\mathrm{f}}c_{\mathrm{f}}T_{\mathrm{o1}}$
$F'(\tau\alpha)_{\mathrm{en}}K_{\tau\alpha\mathrm{b}}(\theta)G_{\mathrm{b}}$
$F'(\tau\alpha)_{\mathrm{en}}K_{\tau\alpha\mathrm{b}}(\theta)G_{\mathrm{b}}$
$F'(\tau\alpha)_{\mathrm{en}}K_{\tau\alpha\mathrm{d}}G_{\mathrm{d}}$
$F'(\tau\alpha)_{\mathrm{en}}K_{\tau\alpha\mathrm{d}}G_{\mathrm{d}}$

Figure 6.3: Diagram of the two compartment model of a solar collector and energy flows. From [2].

equations.

$$\begin{aligned}
dT_{\mathrm{o1}} &= \Big(F'U_0(T_{\mathrm{a}}-T_{\mathrm{f1}}) + n_{\mathrm{c}}c_{\mathrm{f}}Q_{\mathrm{f}}(T_{\mathrm{i}}-T_{\mathrm{o1}}) \\
&\quad + F'(\tau\alpha)_{\mathrm{en}}K_{\tau\alpha\mathrm{b}}(\theta)G_{\mathrm{b}} + F'(\tau\alpha)_{\mathrm{en}}K_{\tau\alpha\mathrm{d}}G_{\mathrm{d}}\Big)\frac{2}{(mC)_{\mathrm{e}}}dt + \sigma_1 d\omega_1 \\
dT_{\mathrm{o2}} &= \Big(F'U_0(T_{\mathrm{a}}-T_{\mathrm{f2}}) + n_{\mathrm{c}}c_{\mathrm{f}}Q_{\mathrm{f}}(T_{\mathrm{o1}}-T_{\mathrm{o2}}) \\
&\quad + F'(\tau\alpha)_{\mathrm{en}}K_{\tau\alpha\mathrm{b}}(\theta)G_{\mathrm{b}} + F'(\tau\alpha)_{\mathrm{en}}K_{\tau\alpha\mathrm{d}}G_{\mathrm{d}}\Big)\frac{2}{(mC)_{\mathrm{e}}}dt + \sigma_2 d\omega_2 \\
&\vdots \\
dT_{\mathrm{o}n_{\mathrm{c}}} &= \Big(F'U_0(T_{\mathrm{a}}-T_{\mathrm{f}n_{\mathrm{c}}}) + n_{\mathrm{c}}c_{\mathrm{f}}Q_{\mathrm{f}}(T_{\mathrm{o}(n_{\mathrm{c}}-1)}-T_{\mathrm{o}n_{\mathrm{c}}}) \\
&\quad + F'(\tau\alpha)_{\mathrm{en}}K_{\tau\alpha\mathrm{b}}(\theta)G_{\mathrm{b}} + F'(\tau\alpha)_{\mathrm{en}}K_{\tau\alpha\mathrm{d}}G_{\mathrm{d}}\Big)\frac{2}{(mC)_{\mathrm{e}}}dt + \sigma_2 d\omega_2
\end{aligned} \tag{6.5}$$

and the measurement equation is

$$Y_k = T_{\mathrm{o}n_{\mathrm{c}}k} + e_k. \tag{6.6}$$

The details are well described in [2].

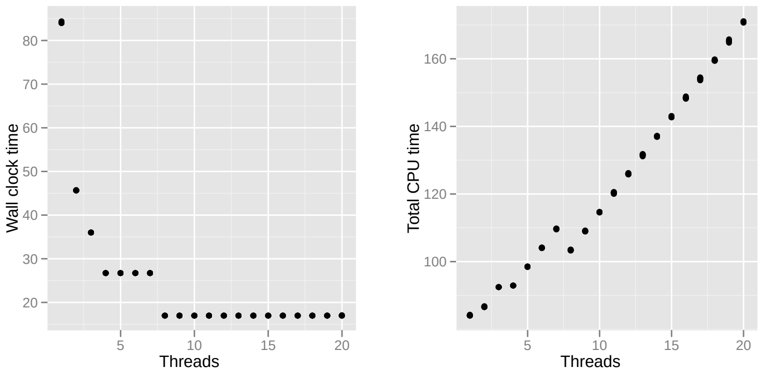
6.2.1 One compartment model

For one compartment the model has one state and one measurement equations. Listing 10 is the corresponding implementation.

```
# Create a NL model
model <- nlctsm$new()
# Add the growth equation
model$addequation(Ktab==(1-b*(1/cosT-1)) *
                        (1/(1+exp(-1000*(cosT-1/(1/b+1))))))
# Add the state equations
model$addstate(To==(Ufa*(Ta-(To+Ti)/2) + Q*c*(Ti-To) +
                    a*A*Ktab*Ib + a*A*Ktad*Id)/Cf)
# Add diffusion terms
model$adddiffusion(1,exp(p22))
# Add the measurement equations
model$addmeas(y~To)
# And the noise terms
model$addnoise(1,exp(e11))
```

Listing 10: CtsmR implementation of the one compartment solar collector model

The initial values are configured as in listing 7 on page 23. The values are taken from the saved CTSM23 model file. Two out of nine parameters are fixed leaving leaving 7 parameters and the initial state as free parameters. The estimation is repeated 10 times for each thread between 1 and 20.



(a) Wall clock (b) Total CPU time

Figure 6.4: Times for the one compartment model

Figure 6.4 shows the fastest estimation time is achieved at 8 cores as expected. The speed-up here is just below 5x. 19 additional seconds are spend on CPU time at 8 cores. This is a 22.6% increase. Again the model is quite quickly estimated.

	CTSM	CtsmR
Objective function	-4487.60124	-4487.60124
Iterations	26	29
Function evaluations	41	39

Table 6.4: Optimisation Results

	CTSM		CtsmR	
Name	Estimate	Std. dev.	Estimate	Std. dev.
To0	$6.341\,10 \times 10^{1}$	$7.324\,70 \times 10^{-2}$	$6.341\,09 \times 10^{1}$	$6.460\,99 \times 10^{-2}$
b	$1.340\,80 \times 10^{-1}$	$1.566\,90 \times 10^{-3}$	$1.340\,79 \times 10^{-1}$	$1.745\,40 \times 10^{-3}$
Ufa	$3.676\,50 \times 10^{1}$	$8.110\,20 \times 10^{-2}$	$3.676\,53 \times 10^{1}$	$8.314\,25 \times 10^{-2}$
c	$4.183\,00 \times 10^{3}$		$4.183\,00 \times 10^{3}$	
a	$8.417\,70 \times 10^{-1}$	$8.054\,60 \times 10^{-4}$	$8.417\,70 \times 10^{-1}$	$8.603\,05 \times 10^{-4}$
A	$1.253\,00 \times 10^{1}$		$1.253\,00 \times 10^{1}$	
Ktad	$9.449\,80 \times 10^{-1}$	$2.299\,80 \times 10^{-3}$	$9.449\,77 \times 10^{-1}$	$2.645\,70 \times 10^{-3}$
Cf	$6.422\,80 \times 10^{4}$	$3.098\,90 \times 10^{2}$	$6.422\,74 \times 10^{4}$	$2.895\,11 \times 10^{2}$
p22	$-1.447\,00 \times 10^{1}$	$8.627\,10 \times 10^{-2}$	$-1.446\,38 \times 10^{1}$	$8.583\,60 \times 10^{-2}$
e11	$-4.275\,20$	$2.110\,80 \times 10^{-2}$	$-4.275\,19$	$2.000\,09 \times 10^{-2}$

Table 6.5: Estimation Results

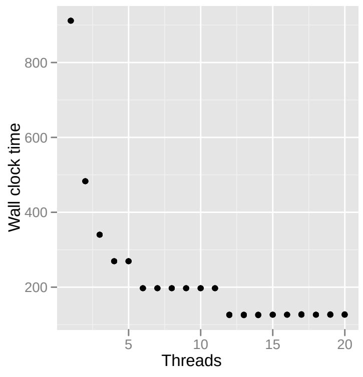
6.2.2 Three compartment model

The expanded three compartment model is shown implemented in CtsmR in listing 11. The initial values are taken from the corresponding CTSM23

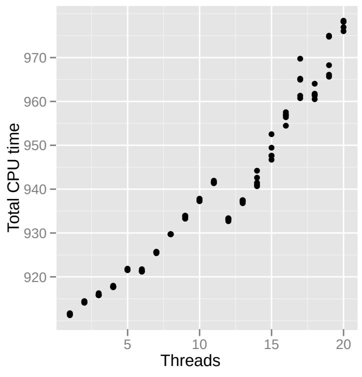
```
# Create a NL model
model <- nlctsm$new()
# Add the growth equation
model$addequation(Pb==a*A/3*(1-b*(1/cosT-1))
                        * (1/(1+exp(-1000*(cosT-1/(1/b+1)))))*Ib)
model$addequation(Pd==a*A/3*Ktad*Id)
model$addequation(deltaT1==Ta - (To1+Ti)/2)
model$addequation(deltaT2==Ta - (To2+To1)/2)
model$addequation(deltaT3==Ta - (To3+To2)/2)
# Add the state equations
model$addstate(To1==Ufa*deltaT1/Cf + Q*c*(Ti-To1)/Cf + Pb/Cf + Pd/Cf)
model$addstate(To2==Ufa*deltaT2/Cf + Q*c*(To1-To2)/Cf + Pb/Cf + Pd/Cf)
model$addstate(To3==Ufa*deltaT3/Cf + Q*c*(To2-To3)/Cf + Pb/Cf + Pd/Cf)
# Add diffusion terms
model$adddiffusion(1,exp(p11),5,exp(p22),9,exp(p33))
# Add the measurement equations
model$addmeas(y==To3)
# And the noise terms
model$addnoise(1,exp(e11))
```

Listing 11: CtsmR implementation of the three compartment solar collector model

file. Figure 6.5 shows a dip in estimation time at 12 which is also the number of free parameters.



(a) Wall clock



(b) Total CPU time

Figure 6.5: Times for the three compartment model

The estimation repeated 5 times and all 100 estimations were identical.

Tables 6.6 and 6.7 on the next page shows the estimates are similar.

	CTSM	CtsmR
Objective function	-12657.52963863876	-12657.52964688
Iterations	44	52
Function evaluations	74	86

Table 6.6: Optimisation Results

	CTSM		CtsmR	
Name	Estimate	Std. dev.	Estimate	Std. dev.
To10	$6.432\,50 \times 10^{1}$	$3.430\,60 \times 10^{-2}$	$6.432\,31 \times 10^{1}$	$8.940\,85 \times 10^{-2}$
To20	$6.399\,20 \times 10^{1}$	$8.124\,70 \times 10^{-2}$	$6.399\,23 \times 10^{1}$	$2.160\,26 \times 10^{-1}$
To30	$6.348\,90 \times 10^{1}$	$1.856\,40 \times 10^{-2}$	$6.348\,89 \times 10^{1}$	$2.838\,30 \times 10^{-2}$
a	$7.974\,50 \times 10^{-1}$	$1.773\,10 \times 10^{-3}$	$7.974\,49 \times 10^{-1}$	$3.406\,17 \times 10^{-3}$
A	$1.253\,00 \times 10^{1}$		$1.253\,00 \times 10^{1}$	
b	$1.542\,90 \times 10^{-1}$	$2.332\,90 \times 10^{-3}$	$1.542\,81 \times 10^{-1}$	$3.235\,38 \times 10^{-3}$
Ktad	$9.590\,40 \times 10^{-1}$	$5.203\,00 \times 10^{-3}$	$9.590\,42 \times 10^{-1}$	$9.682\,13 \times 10^{-3}$
Ufa	$1.162\,40 \times 10^{1}$	$8.420\,70 \times 10^{-2}$	$1.162\,42 \times 10^{1}$	$8.290\,04 \times 10^{-2}$
Cf	$2.854\,40 \times 10^{4}$	$1.543\,20 \times 10^{2}$	$2.854\,36 \times 10^{4}$	$2.356\,85 \times 10^{2}$
c	$3.970\,00 \times 10^{3}$		$3.970\,00 \times 10^{3}$	
p11	$-2.914\,50$	$3.212\,50 \times 10^{-2}$	$-2.914\,57$	$3.157\,43 \times 10^{-2}$
p22	$-3.594\,60$	$3.262\,50 \times 10^{-2}$	$-3.594\,63$	$3.749\,85 \times 10^{-2}$
p33	$-2.012\,60 \times 10^{1}$	$1.221\,90 \times 10^{-1}$	$-1.975\,70 \times 10^{1}$	$1.126\,58 \times 10^{-1}$
e11	$-9.658\,80$	$8.710\,20 \times 10^{-2}$	$-9.658\,74$	$8.140\,28 \times 10^{-2}$

Table 6.7: Estimation Results

6.3 Glucose concentration model

This model is based on [11, 27]. The CtsmR implementation here is based on the work by Anne Katrine Duun-Henriksen.

This model join the CtsmR test arsenal rather late. What made it interesting is it size: 10 states and and 28 parameters. The optimisation space has a full space of 38 dimensions. Listing 12 provides the CtsmR implementation.

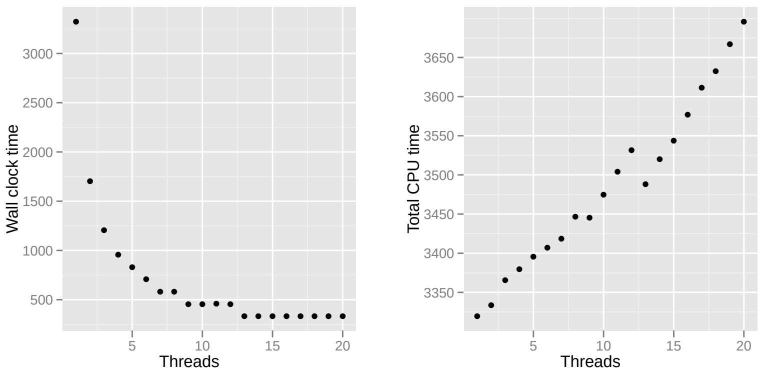
```
# Create a NL model
model <- nlctsm$new()
# Add the equations
model$addequation(FR==(0.003*(Q1/VG-9)*VG)/(1+exp(ggf*(9-Q1/VG))))
model$addequation(F01c==(F01*(Q1/VG)/4.5)/(1+exp(-ggc*(4.5-(Q1/VG))))+
                  F01/(1+exp(ggc*(4.5-(Q1/VG)))))
# Add the state equations
model$addstate(Q1==D2/tauD-F01c-FR-x1*Q1+k12*Q2+EGP0*(1-x3))
model$addstate(Q2==x1*Q1-(k12+x2)*Q2)
model$addstate(x1==-ka1*x1+kb1*I)
model$addstate(x2==-ka2*x2+kb2*I)
model$addstate(x3==-ka3*x3+kb3*I)
model$addstate(I==(S2/tauS)/VI-I*ke)
model$addstate(D1==Ag*d-D1/tauD)
model$addstate(D2==D1/tauD-D2/tauD)
model$addstate(S1==Usc-S1/tauS)
model$addstate(S2==S1/tauS-S2/tauS)
# Add diffusion terms
model$adddiffusion(1,exp(s1),12,exp(s2),23,exp(s3),34,exp(s4),45,exp(s5),
                  56,exp(s6),67,exp(s7),78,exp(s8),89,exp(s9),100,exp(s10))
# Add the measurement equations
model$addmeas(Gout==Q1/VG)
# And the noise terms
model$addnoise(1,sG)
```

Listing 12: CtsmR implementation of the Hovorka model

This model is estimated using simulated data. All states have been fixed plus another two parameters. Thus 26 parameters must be estimated. The servers at the G-bar only have 24 cores so all directions for the gradient cannot be estimated simultaneously. The best performance is expected at 13 cores. That is because 13 is the highest number dividing 26.

The estimation is slow so there is only one estimation per number of cores. Figure 6.6 on the next page shows the resulting times. The fastest estimation time is found at 13. This is where each core is essentially calculating both points while approximating the gradient. At 13 cores the speed-up is 10x and relative additional CPU cost is 8.8%.

The estimates are once again very similar. The results are not shown due to the 26 parameters.



(a) Wall clock (b) Total CPU time

Figure 6.6: Times for the Hovorka model

6.4 Marine Ecosystem in Skive Fjord

The two models tested here were provided by Jan Kloppenborg Møller [17]. His paper is a methodology description with an application to a marine ecosystem in Skive Fjord. The change of water column nitrogen and phytoplankton nitrogen is modelled by a SDE. Contrary to any of the previous model eq. (6.7) has state dependent diffusion.

$$
\begin{aligned}
d\begin{bmatrix} X_{w,t} \\ X_{p,t} \end{bmatrix} = \begin{bmatrix} N_{ex,t}Q_t \\ 0 \end{bmatrix} dt + \begin{bmatrix} -Q_t - a_{wp} - a_{wt} & a_{pw} \\ a_{pw} & -a_{pw} - Q_t \end{bmatrix} \begin{bmatrix} X_{w,t} \\ X_{p,t} \end{bmatrix} dt \\
+ \begin{bmatrix} \sigma_w X_{w,t} & 0 \\ 0 & \sigma_p X_{p,t} \end{bmatrix} \begin{bmatrix} 1 & r_{12} \\ r_{12} & 1 \end{bmatrix} d\boldsymbol{w}_t
\end{aligned}
\tag{6.7}
$$

Equation (6.7) does not comply with the requirements of CTSM and its use of Kalman filtering due to the state dependent diffusion. Through a Lamperti transformation the state dependence is removed and the model can be estimated using CTSM. For eq. (6.7) the Lamperti transformation is simply a log transformation [17, p. 10].

This particular dataset has a lot of missing observations. The sampling time of the different inputs vary a lot, but the dataset has a set of observations per day. As a result three of the inputs are missing 92% of the observations as the sampling time was high.

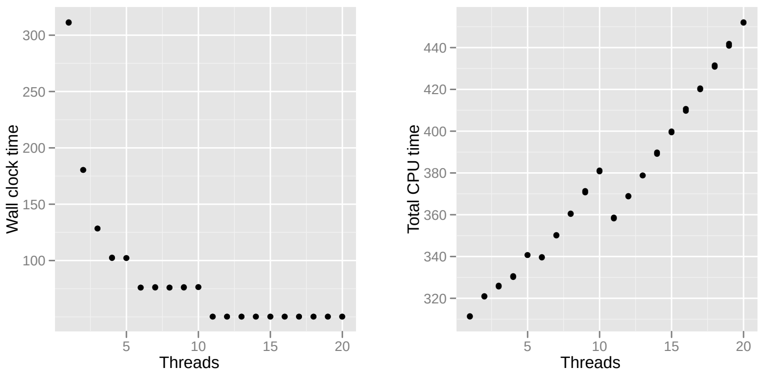
6.4.1 Basic Marine model

Equation (6.7) on the preceding page was only tried at a very late stage. A larger extended version of the model was difficult to get working in CtsmR. Hoping for an uncaught implementation error this model was tried. Although the model is smaller the dataset remains the same. There are 11

```
# Create a NL model
model <- nlctsm$new()
# Add the equations
model$addequation(Xw==exp(sxw*Zw))
model$addequation(Xp==cp*exp(sxp*Zp))
model$addequation(f==1)
# Add the state equations
model$addstate(Zw==(Nex*Q/Xw-awl-Q-awp*f+apw*Xp/Xw)/sxw-sxw*(1+r^2)/2)
model$addstate(Zp==(awp*Xw/Xp*f-apw-Q-apl)/sxp-sxp*(1+r^2)/2)
# Add diffusion terms
model$adddiffusion(1,1,2,r,3,r,4,1)
# Add the measurement equations
model$addmeas(Yw==log(Xw+Xp))
model$addmeas(Yp==log(cp)+sxp*Zp)
model$addmeas(Ypri==sxw*Zw+log(awp)+log(f))
# And the noise terms
model$addnoise(1,sw,5,sp,9,spr)
```

Listing 13: CtsmR implementation of the eq. (6.7)

free parameters and the fastest estimation time is at 11 cores illustrated in fig. 6.7.



(a) Wall clock (b) Total CPU time

Figure 6.7: Times for small model of Skive Fjord

The speed-up is 4.4x for an additional charge of 15x. Table 6.8 on the next page shows the results which again is near identical.

	CTSM	CtsmR
Objective function	1445.753337696099	1445.7533376961
Iterations	49	56
Function evaluations	60	71

Table 6.8: Optimisation Results

	CTSM		CtsmR	
Name	Estimate	Std. dev.	Estimate	Std. dev.
Zw0	$-6.771\,00$	$4.493\,30$	$-6.771\,05$	$4.439\,77$
Zp0	$-2.346\,20$	$1.393\,60$	$-2.346\,23$	$1.234\,25$
sxw	$6.115\,30 \times 10^{-2}$	$3.995\,40 \times 10^{-3}$	$6.115\,34 \times 10^{-2}$	$3.669\,79 \times 10^{-3}$
cp	1		1	
sxp	$6.246\,00 \times 10^{-1}$	$8.486\,50 \times 10^{-2}$	$6.246\,01 \times 10^{-1}$	$8.146\,71 \times 10^{-2}$
awl	$7.031\,70 \times 10^{-3}$	$3.282\,10 \times 10^{-3}$	$7.031\,70 \times 10^{-3}$	$3.106\,89 \times 10^{-3}$
awp	$1.589\,50 \times 10^{-2}$	$2.209\,40 \times 10^{-3}$	$1.589\,51 \times 10^{-2}$	$2.177\,43 \times 10^{-3}$
apw	$5.264\,10 \times 10^{-2}$	$3.406\,30 \times 10^{-2}$	$5.264\,05 \times 10^{-2}$	$3.117\,88 \times 10^{-2}$
r	$-1.644\,90 \times 10^{-1}$	$4.648\,70 \times 10^{-2}$	$-1.644\,95 \times 10^{-1}$	$4.377\,29 \times 10^{-2}$
apl	0		0	
sw	$1.348\,50 \times 10^{-2}$	$2.448\,60 \times 10^{-3}$	$1.348\,47 \times 10^{-2}$	$2.262\,69 \times 10^{-3}$
sp	$3.039\,00 \times 10^{-1}$	$1.481\,90 \times 10^{-1}$	$3.038\,97 \times 10^{-1}$	$1.449\,88 \times 10^{-1}$
spr	$3.545\,90$	$2.849\,30 \times 10^{-1}$	$3.545\,94$	$2.872\,63 \times 10^{-1}$

Table 6.9: Estimation Results

6.4.2 Extended Marine model

The model shown in eq. (6.7) on page 32 is the basic model which is extended through out the paper. This is the model from [17, p. 18] where the diffusion terms have been changed. The model is Lamperti transformed to get state independent diffusion. Furthermore one of the parameters is added as a state only with a diffusion term.

The R implementation is given in listing 14 on the next page.

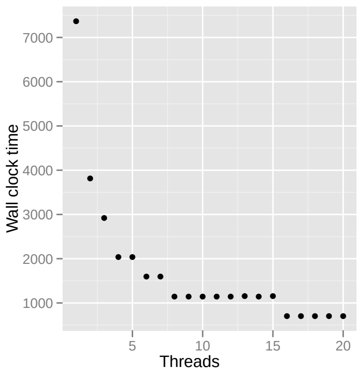
Figure 6.8 on the facing page are based on a single estimation per cores used.

The estimation is very time consuming spending 7500 seconds on the estimation using a single core. Running at the optimal 16 cores resulted in a 10.4x speed-up for an additional CPU charge of 3.3%.

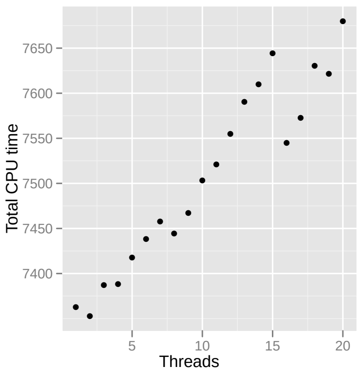
The estimates are once again very similar. The results are not shown due to the number of parameters.

```
# Create a NL model
model <- nlctsm$new()
# Add the equations
model$addequation(Xw==(sxw*(1-gw)*Zw)^(1/(1-gw)))
model$addequation(Xp==(sxp*(1-gp)*Zp)^(1/(1-gp)))
model$addequation(f==Xp*(gr/(kgr+gr))/(kw+Xw))
model$addequation(dpw==Xw^(-gw)/sxw)
model$addequation(dpp==Xp^(-gp)/sxp)
model$addequation(corw==Xw^(gw-1)*gw*(1+r^2)*sxw/2)
model$addequation(corp==Xp^(gp-1)*gp*(1+r^2)*sxp/2)
# Add the state equations
model$addstate(Zw==dpw*(Nex*Q-(awl+Q+exp(awp)*f)*Xw+apw*Xp)-corw)
model$addstate(Zp==dpp*(ap0+exp(awp)*Xw*f-(apw+Q)*Xp)-corp)
model$addstate(awp==0)
# Add diffusion terms
model$adddiffusion(1,1,2,r,4,r,5,1,9,0)
# Add the measurement equations
model$addmeas(Yw==log(Xw+Xp))
model$addmeas(Yp==log(Xp))
model$addmeas(Ypri==log(Xw)+awp+log(f))
# And the noise terms
model$addnoise(1,sw,5,sp,9,spr)
```

Listing 14: CtsmR implementation of the extended marine eco model



(a) Wall clock



(b) Total CPU time

Figure 6.8: Times for the large model of Skive Fjord

CHAPTER 7

Discussion

All test models were estimated as expected. That is the estimates from CtsmR as highly similar to those of CTSM23. The observed differences is explained in section 7.1.

The primary observation was that the estimation of any valid CTSM model is always fastest when the used number of cores equals the number of free parameters. This is not at all surprising in theory, but is confirmed here. The speed-up is determined in general in section 7.2 on the following page.

All models tested here were already working in CTSM23. It was a matter of reproducing the results. However during model identification a new feature of CtsmR can help diagnosing when an estimation does not work. More on this in section 7.3 on page 41.

7.1 The discrepancies

All the parameter estimations shown in chapter 6 on page 21 vary somewhat from the previous results. All the estimations done during this thesis were done on the same 64 bit AMD architecture running Linux. The results will vary across 32/64 bit, compiler suites as well as the operating system. This is however not the case here.

The Jacobian in CTSM23 is determined using automatic differentiation. In CtsmR it is determined analytically, but may or may not be its most simplified form. The fed-batch model in section 6.1 on page 21 was estimated in CTSM23 again. When the optimisation finished the four jacobians were replaced with those derived analytically in CtsmR. After recompiling the source code CTSM23 was asked for one more estimation. The results are shown in table 7.1 on the following page. Some "larger" changes are highlighted.

	CTSM23 AD		CtsmR diff	
Name	Estimate	Std. dev.	Estimate	Std. dev.
X0	1.009 55	$1.059\,68 \times 10^{-2}$	1.009 55	$1.057\,13 \times 10^{-2}$
S0	$2.383\,47 \times 10^{-1}$	$9.308\,00 \times 10^{-3}$	$2.383\,47 \times 10^{-1}$	$9.058\,98 \times 10^{-3}$
V0	1.003 95	$7.917\,15 \times 10^{-3}$	1.003 95	$7.365\,02 \times 10^{-3}$
mumax	1.002 24	$2.843\,86 \times 10^{-3}$	1.002 24	$4.633\,68 \times 10^{-3}$
K2	5×10^{-1}		5×10^{-1}	
K1	$3.162\,94 \times 10^{-2}$	$1.313\,94 \times 10^{-3}$	$3.162\,93 \times 10^{-2}$	$1.932\,08 \times 10^{-3}$
Y	5×10^{-1}		5×10^{-1}	
SF	$1.000\,00 \times 10^{1}$		$1.000\,00 \times 10^{1}$	
sig11	$1.552\,98 \times 10^{-27}$	$7.553\,69 \times 10^{-26}$	$1.957\,71 \times 10^{-27}$	$6.153\,47 \times 10^{-26}$
sig22	$1.765\,42 \times 10^{-6}$	$1.330\,94 \times 10^{-5}$	$8.466\,89 \times 10^{-7}$	$4.435\,86 \times 10^{-6}$
sig33	$1.149\,93 \times 10^{-8}$	$1.268\,31 \times 10^{-7}$	$1.087\,10 \times 10^{-8}$	$7.805\,00 \times 10^{-8}$
s11	$7.524\,75 \times 10^{-3}$	$9.997\,02 \times 10^{-4}$	$7.524\,80 \times 10^{-3}$	$1.036\,89 \times 10^{-3}$
s22	$1.063\,61 \times 10^{-3}$	$1.383\,74 \times 10^{-4}$	$1.063\,61 \times 10^{-3}$	$1.378\,15 \times 10^{-4}$
s33	$1.138\,85 \times 10^{-2}$	$1.530\,64 \times 10^{-3}$	$1.138\,85 \times 10^{-2}$	$1.554\,92 \times 10^{-3}$

Table 7.1: Estimation Results

The automatic differentiation is using two work vectors which is looped through several times. By replacing the Jacobians the estimation time dropped. Although it remains unmeasured the difference was clear. It it also rather expected as the Jacobian derived symbolically are simple calculations without any use of loops and work vectors.

Order of parameters CTSM23 and CtsmR vary in the way the parameters are found in the equations. In CtsmR the order is: Initial states, sorted lists of parameters in the drift and *dt* parts, diffusion specific parameters and noise specific parameters. In the fed-batch example three parameters have swapped place. By forcing CtsmR to use the same order of parameters as CTSM23 the results become identical to those from CTSM23 using the analytically derived Jacobians. The order of parameters is rather random in CTSM as they are listed as they appear in the equations. One order is not better than another per se, but CtsmR will not be affected should a user swap the order of the equations in a model.

7.2 Speed-up

All the wall clock profiles figs. 6.1 and 6.4 to 6.7 on pages 24–33 are similar in shape. The difference is the number of free parameters in the model. The highest speed-up is always achieved using the same number of cores as free parameters. The speed-ups is a relative measure and it seems natural that the speed-up depends on the amount of free parameters. Adding one extra parameter calls for two additional evaluations of the objective function. To study this relationship all models went through the following scheme.

- Fix all parameters to either the initial or optimal values
- Release one parameter and set the original bounds
- Estimate the model using one core
- Estimate the model using optimal number of cores
- Repeat until all parameters have been set free

The estimation of some of the configurations did as the ODE solution would fail. Most models are complicated thus a bad initiation can be problematic. To actually perform this is very simple in CtsmR due to the ability to script. First the models were estimated as they were originally intended and the results were subsequently used as described in section 7.2 on the preceding page. A nice ability which is be very cumbersome in CTSM23.

In practice all models were tried using both initialisations. The model with a single free parameter was obviously not estimated as the speed-up measure is 1. Parameters originally fixed remained fixed during this test and the estimation was skipped. Finally the info flag was check for both the single and parallel runs. This will not exclude all degenerative/troubled estimations. In a number of cases the forward propagation ODE is so stiff that the integration exceeds the allow steps. This is followed by a warning printed to R. CTSM will restart the integration 1000 times each followed by a warning. The same will of cause happen running in parallel but the printing of the warnings to R may increase the overhead disproportionally thus lowering the speed-up measure.

Figure 7.1 on the following page shows the relative speed-up measures for the test models plotted against the number of free parameters. For every model a speed-up of 1 for one free parameter has been added.

Figure 7.1 on the next page shows a clear and expected trend. The speed-up is not identical to the free parameters as expected. A linear fit is done in R by fixing the $(1,1)$ point. This is reasonable as the speed-up for one free parameter is simply 1. Fitting eq. (7.1) using `lm()` in R gives a slope of $a = 0.5716$. Thus eq. (7.2) is the *rule of thumb* for the speed-up in CtsmR in parallel.

$$\mathrm{SU} - 1 = a \cdot (\#\mathrm{pars} - 1) \tag{7.1}$$

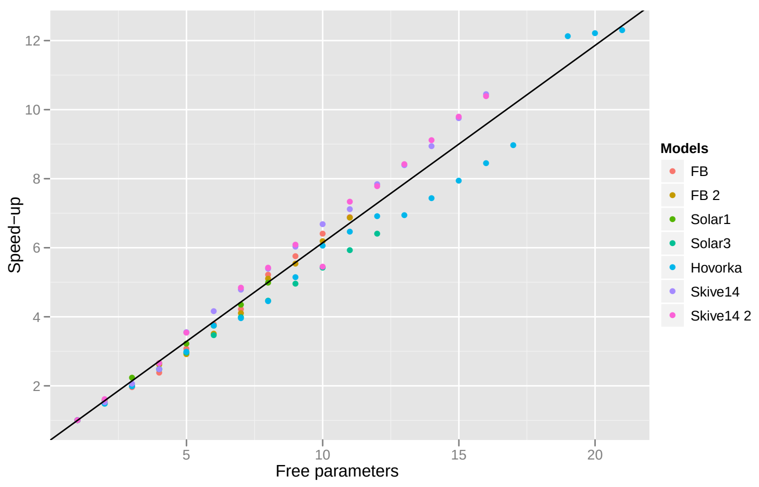


Figure 7.1: Speed-up by number of free parameters. *FB* 2 and *Skive14 2* were initiated using optimal values. Rest using those from the original CTSM23 files.

$$\text{SU} = 0.5716 \cdot \#\text{pars} + 0.4284 \tag{7.2}$$

7.2.1 Additional CPU cost

The optimisation algorithm itself is serial. Determining the gradient is a truly expensive task whereas the remaining matrix vector multiplications finding the search direction and BFGS updates are very cheap in comparison. During the serial sections all but one core are idle and wasting CPU time. However the complexity of the gradient compared to the complexity of the Quasi-Newton algorithm itself is rather high.

Figure 7.2 on the facing page shows increased total CPU time as high as 60%. However this is the fed-batch model which is very fast to estimate. As the complexity and thus estimation time increases the relative additional CPU cost decreases. The time consuming extended marine eco model only charges 3.3% extra CPU time to achieve a 10.4x speed-up.

7.3 Diagnostics

CTSM23 prints the current point in the parameter space during the estimation. The values are overwritten at some rate which does not follow the rate of iteration. In reality obtaining a trace in CTSM23 is not possible.

The codebase of CTSM is now for each iteration saving

- the diagonal elements of the Hessian

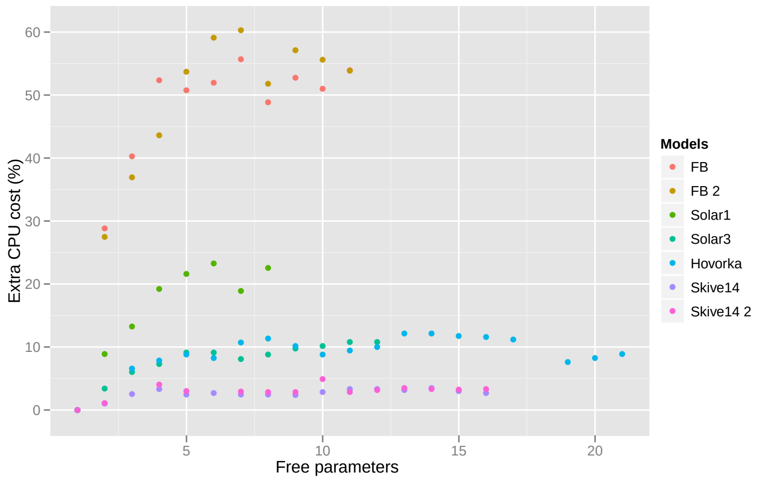


Figure 7.2: Additional CPU time in percent

- the gradient
- the step length
- the free parameters
- the negative log-likelihood

This information can shed some light on what the optimiser is doing. When the optimisation does not converge it can be useful to diagnose which parameter is problematic.

The trace of the parameters of the optimisation of the extended marine eco model from section 6.4.2 on page 34 is shown in fig. 7.3 on the following page. The trace is also showing that all the estimates stabilises at the end thus indicating convergence to some minimum. Figure 7.4 on the next page is the negative log-likelihood. It is decreasing well and flattens out indicating a local minimum.

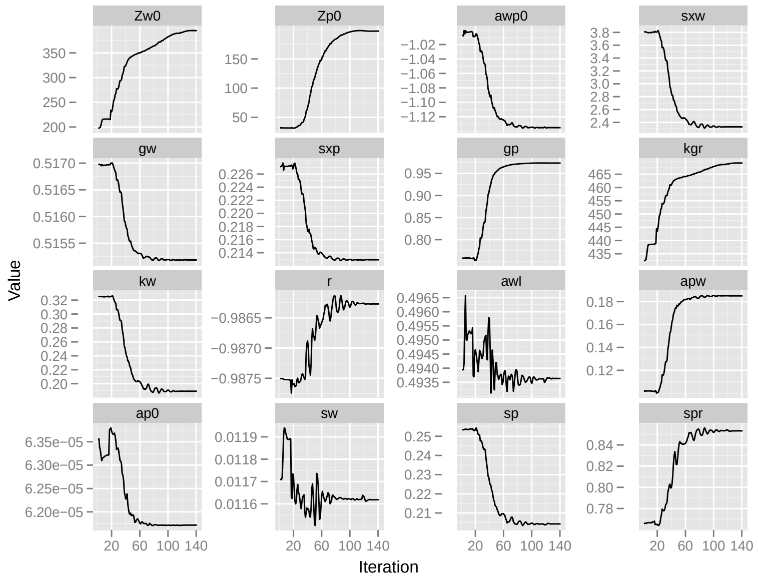


Figure 7.3: Back transformed parameter trace from the extended marine eco model in section 6.4.2

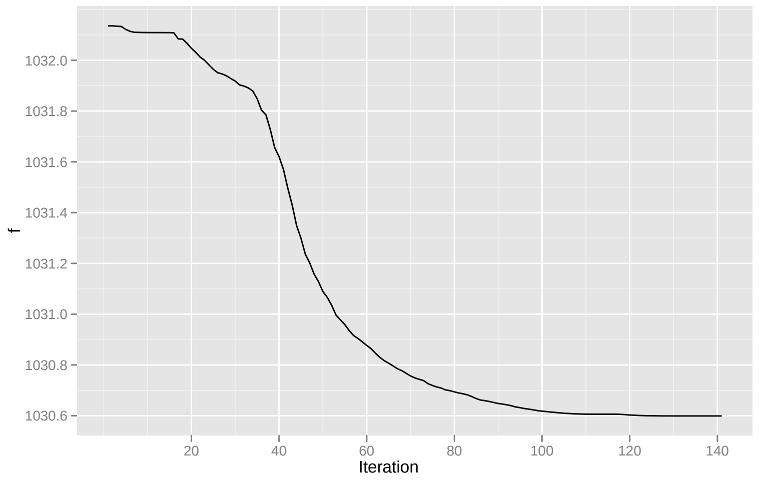


Figure 7.4: Trace of the negative log-likelihood from the extended marine eco model in section 6.4.2

CHAPTER 8

Future Development

CtsmR is an almost ready to use R package. However many ideas for the further development have emerged.

8.1 Polishing the package

The package still contains some rough edges which I will upon finishing this thesis will take care off.

CRAN or *The Comprehensive R Archive Network* is the major host of packages contributed by the community. Uploading CtsmR to CRAN is the next logical step.

Documentation of the classes and exposed functions is missing. This is a (fair) requirement before uploading to CRAN. Hadley Wickham's Roxygen2 will be used.

Unit testing is a great way to set up tests to ensure the correctness of the code. Bad releases should not be possible. Matthias Burger's RUnit package should be used.

8.2 Optimisers

Currently it is the well-known unconstrained BFGS Quasi-Newton doing the optimiser. The parameter space in the CTSM models are usually bounded. Thus the constrained problem is first transformed into an unconstrained problem before the optimisation is begun.

The current optimiser it the `VA13` from the Harwell Subroutine Library. The HSL Mathematical Software Library does not provide non-linear optimisation tools any longer. They are now developed in the GALAHAD library which is a thread safe Fortran library for both unconstrained and bound-constrained problems [8].

8.2.1 Stochastic Optimisation

The most time consuming task here is evaluating the loss function. The parameter space does have to be very high dimensional before the CTSM problems consume time. Computing the gradient is means varying all parameters twice. Adaptive Simultaneous Permutation Stochastic Approximation is a stochastic analogue of the Newton-Raphson methods [24]. The adaptive SPSA is estimating the Hessian and gradient. The main recursions are

$$\hat{\boldsymbol{\theta}}_{k+1} = \hat{\boldsymbol{\theta}}_k - a_k \overline{\overline{\boldsymbol{H}}}_k^{-1} \boldsymbol{G}_k(\hat{\boldsymbol{\theta}}_k), \quad \overline{\overline{\boldsymbol{H}}}_k = f_k(\overline{\boldsymbol{H}}_k) \tag{8.1}$$

$$\overline{\boldsymbol{H}}_k = \frac{k}{k+1}\overline{\boldsymbol{H}}_{k-1} + \frac{1}{k+1}\hat{\boldsymbol{H}}_k, \quad k = 0, 1, 2, \ldots \tag{8.2}$$

$\boldsymbol{G}_k$ is a gradient approximation, $\hat{\boldsymbol{H}}_k$ is the k-th iterate estimate of the Hessian and $\boldsymbol{f}(\cdot)$ is some function ensuring the Hessian is positive definite. The a_k is a non negative gain like the step length in the deterministic methods. The clever part is how the gradient and Hessian are estimated. The gradient requires two evaluations of the loss function. These two points is based on perturbing the current point with independent Bernoulli ± 1 in each dimension. The spacing is determined by a iteration dependent gain factor. The Hessian requires another two evaluations which are further perturbation of those two points evolved in the gradient. Again the perturbation is random and the spacing is controlled by another gain factor. The details are omitted here but are found in [24, 26].

Per iteration the adaptive SPSA only requires 4 evaluations of the loss function independent of the dimension. This should really help speeding up some of the high dimensional problems. [25] suggests a method using feedback and weighting of the Hessian to get faster convergence than regular SPSA.

One issue is the gain factors. Some are suggested in [24] while some are problem dependent and rely on expert judgement. If some heuristics cannot be determined to get a true black box method then these are just additional settings.

8.3 Fortran 90/95

Fortran 77 is an old language. Since Fortran 77 there have been a number of update to the standard; Fortran 90, 95, 2003 and 2008. Fortran 90 was a major revision and brought many interesting new features which will simply the CTSM Fortran codebase.

Masked array assignment Throughout the code there are many evaluations of the user defined matrices. These are often save in other variables. Some matrices are augmented by a row when evaluated, thus a subset of the matrix is saved elsewhere. Currently this is done through nested loops. Fortran 90 allows `X(1:N)=R(1:N)`.

Dynamic memory allocation Fortran 77 does have assumed size variables, but is otherwise not able to dynamically allocate memory on the heap. Fortran 90 introduced an `ALLOCATABLE` attributed. When needed memory can now be allocated through `ALLOCATE`.

The current code depends on variable length variables in a common block. This is impossible in Fortran 77. Thus the size must be known at compile time which makes it impossible to have a compiled library working for unknown problems. The can be solved using by using a module (also new) for allocatable variables.

Operator overloading Although R is performing the differentiation analytically now the previous method was to use automatic differentiation through source code transformation. Operator overloading is another strategy for automatic differentiation and there are several such tools available[1].

Other useful things `DO` loops are terminated by a label. This means there are a lot of labels. Although some supported it already `END DO` are now standard by Fortran 90.

Free form input allows more flexibility in the source code layout. The layout has been freed so to speak.

Functions and variables are allowed to be 31 characters long.

8.3.1 Portability

Portability is always a concern. We want CTSM to be functional across platforms. R is open source and as such written to be compiled with GNU's compilers. It can however be compiled with Intel's and Oracle's compiler suites under Linux [22, pp. 51-52].

GNU's Fortran compiler is `gfortran` and it is a Fortran 95/2003/2008 compiler[6]. It offers a `-std=legacy` compiler option which suppresses warnings related to Fortran 77 code.

Oracle Studio's Fortran compiler is a Fortran 95 compiler [20, p. 177]. It will compile standard conforming Fortran 77 for backward compatibility. Version 12.1 (from Sun's time) is available on the Gbar at DTU.

Intel's `ifort` does compiles Fortran 77 code and provides backward support for Fortran 77 specifics [12]. The Intel compiler suite is not available at DTU.

All the compilers our users are likely to use will compile Fortran 90/95 code. In fact all 3 compilers only support Fortran 77 for backward compatibility. There might however be problems with very old operating system which will not compile anything but Fortran 77 [23, p. 23]. In general these are irrelevant concerns for CTSM.

8.4 GPU Acceleration

Exploiting the graphical processing units (or GPU) is getting more and more popular. GPU's have hundreds of cores and can bring supercomputers to workstations. NVIDIA's Tesla C2075 has 448 CUDA cores and 6 GB of memory [19]. Each core clock at 1150 MHz.

While the problems where CTSM is applied it not all that high dimensional in parameter space there are many calculations which could be done in parallel. The biggest model I have seen tried in CTSM has some 50 parameters and 10 states. The central finite difference approximation of the gradient is $2N$ independent calculations. Solving the forward Kolmogorov difference equation for the states and variance is N and $\frac{N(N+1)}{2}$ coupled ODEs respectively. To fully exploit this requires new tools for solving coupled ODEs in parallel.

The clock frequency of a GPU core is much lower than a CPU. Modern desktop CPUs have 2-6 cores working at frequencies around 3 GHz. The servers at the Gbar, DTU have 2x12 cores AMD 6168 CPUs. Each core is working at 1900 MHz [4]. Despite the Gbar is slower per core than a regular laptop is will run CTSM problems with 2-3 or more parameters faster in parallel.

8.5 Extending the R interface

The new R interface is the first take. At an internal CTSM user meeting several ideas surfaced.

Model specification trough a graphical user interface written in Tcl/Tk. There are many inputs in these types of models and it can provide a quicker overview having graphical entry point. The graphical interface should be able to write out the corresponding R code. Thus is will serve as a learning tool too.

R has a number of functions like `plot()`, `resid()` and `anova`. Plotting an output from a fitted model should show diagnostic plots like when plotting the output of `lm()`. There are a number of such functions which are logical to implement. This will not only be user friendly but will integrate CTSM further into the "regular" R way of working.

CHAPTER 9

Conclusion

The aim of this thesis was to reproduce the CTSM23 results faster in a flexible environment.

R is a widely used statistical language and is a perfect environment providing the flexibility, reproducibility and prototyping any scientist would and should appreciate. With CtsmR a model can be extended, do multiple estimation on different datasets, sweep of initial parameters, use optimised point as initial values in the next and etc. The ability to script is a much desired feature which soon will be available for everybody to use.

Speed is essential, but the models tend to get more complicated. For some years now the race of the Ghz has been replaced and focus is now turned towards multi core systems. A modern laptop will have two of four cores. Using three cores, leaving one for the system itself, the expected speed up is 2.

Running in parallel is not completely free. The total CPU time will increase using more threads. That said the relative additional CPU time does decrease for increased estimation time. Complicated models gain a substantial speed-up at a low cost.

The product of this thesis is only the beginning of CtsmR and the proposed future developments.

Bibliography

[1] *AD Tools for Fortran95*. autodiff.org. URL: `http://www.autodiff.org/?module=Tools&language=Fortran95` (visited on 10/05/2011) (cit. on p. 45).

[2] P. Bacher, H. Madsen and B. Perers. 'Models of the heat dynamics of solar collectors for performance testing'. In: *Proceedings of ISES Solar World Conference 2011*. 2011 (cit. on p. 26).

[3] *BLAS Frequently Asked Questions (FAQ)*. 2005. URL: `http://netlib.org/blas/faq.html#9` (cit. on p. 7).

[4] DTU G-bar. *Hardware*. URL: `http://www.gbar.dtu.dk/wiki/Hardware#12_HP_ProLiant_SL165z_G7_servers` (visited on 29/10/2011) (cit. on p. 46).

[5] Eclipse. *Eclipse Phortran*. URL: `http://www.eclipse.org/photran/` (cit. on p. 11).

[6] *gfortran — the GNU Fortran compiler, part of GCC*. GNU. URL: `http://gcc.gnu.org/wiki/GFortran` (cit. on p. 46).

[7] GNU. *6.1.16 OpenMP*. URL: `http://gcc.gnu.org/onlinedocs/gfortran/OpenMP.html` (cit. on p. 8).

[8] Nicholas I. M. Gould, Dominique Orban and Philippe L. Toint. 'GALAHAD, a library of thread-safe Fortran 90 packages for large-scale nonlinear optimization'. In: *ACM Trans. Math. Softw.* 29.4 (Dec. 2003), pp. 353–372. ISSN: 0098-3500. DOI: `10.1145/962437.962438`. URL: `http://doi.acm.org.globalproxy.cvt.dk/10.1145/962437.962438` (cit. on p. 44).

[9] A. Griewank. *Jakef*. Argonne National Laboratory. 1988. URL: `http://www.netlib.org/jakef/` (cit. on p. 6).

[10] AC Hindmarsh and RS Stepleman. 'ODEPACK, A Systematized Collection of ODE Solvers , R. S. Stepleman et al. (eds.), North-Holland, Amsterdam, (vol. 1 of), pp. 55-64.' In: *IMACS Transactions on Scientific Computation* 1 (1983), pp. 55–64 (cit. on p. 7).

[11] Roman Hovorka et al. 'Nonlinear model predictive control of glucose concentration in subjects with type 1 diabetes'. In: *Physiological Measurement* 25.4 (Aug. 2004), pp. 905–920. ISSN: 0967-3334. DOI: `10.1088/0967-3334/25/4/010`. URL: `http://iopscience.iop.org.globalproxy.cvt.dk/0967-3334/25/4/010` (cit. on p. 31).

[12] Intel. *Intel® Fortran Compiler User and Reference Guides*. 304970-006US. URL: `http://software.intel.com/sites/products/documentation/hpc/compilerpro/en-us/fortran/lin/main_for_lin.pdf` (cit. on p. 46).

[13] Niels Rode Kristensen and Henrik Madsen. *Continuous Time Stochastic Modelling. CTSM 2.3 - User's Guide*. 10th Dec. 2003 (cit. on p. 22).

[14] Niels Rode Kristensen, Henrik Madsen and Sten Bay JÃ¸rgensen. 'Parameter estimation in stochastic grey-box models'. In: *Automatica* 40.2 (Feb. 2004), pp. 225–237. ISSN: 0005-1098. DOI: `10.1016/j.automatica.2003.10.001`. URL: `http://www.sciencedirect.com/science/article/pii/S000510980300298X` (cit. on p. 1).

[15] *LAPACK 3.1 Release Notes*. 2006. URL: `http://www.netlib.org/lapack/revisions.info.3.1` (cit. on p. 7).

[16] *LAPACK 3.3.0 Release Notes*. University of Tennessee et al. 14th Nov. 2010. URL: `http://www.netlib.org/lapack/lapack-3.3.0.html` (visited on 20/04/2011) (cit. on p. 7).

[17] Jan Kloppenborg Møller, Jacob Carstensen and Henrik Madsen. 'Structural Identification and Velidation in Stochastic Differential Equation based Models - With application to a Marine Ecosystem NP-model'. Submitted. 2011 (cit. on pp. 5, 32, 34).

[18] J. Nocedal and S.J. Wright. *Numerical optimization*. Springer series in operations research. Springer, 2006. ISBN: 9780387303031 (cit. on p. 4).

[19] NVIDIA. *NVIDIA® TESLA^{TM} C2075 COMPANION PROCESSOR*. URL: `http://www.nvidia.com/docs/IO/43395/NV-DS-Tesla-C2075.pdf` (cit. on p. 46).

[20] Oracle. *Oracle® Solaris Studio 12.2: Fortran User's Guide*. 2011. URL: `http://docs.oracle.com/cd/E18659_01/pdf/821-1382.pdf` (cit. on p. 46).

[21] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria, 2011. ISBN: 3-900051-11-9. URL: `http://www.R-project.org/` (cit. on p. 6).

[22] R Development Core Team. *R Installation and Administration*. Version 2.13.2. 30th Sept. 2011. ISBN: 3-900051-11-9. URL: `http://cran.r-project.org/doc/manuals/R-admin.pdf` (cit. on p. 46).

[23] R Development Core Team. *Writing R Extensions*. Version 2.13.2. 30th Sept. 2011. ISBN: 3-900051-11-9. URL: `http://cran.r-project.org/doc/manuals/R-exts.pdf` (cit. on p. 46).

[24] J. C Spall. 'Adaptive stochastic approximation by the simultaneous perturbation method'. In: *IEEE Transactions on Automatic Control* 45.10 (Oct. 2000), pp. 1839–1853. ISSN: 0018-9286. DOI: `10.1109/TAC.2000.880982` (cit. on p. 44).

[25] J. C Spall. 'Feedback and Weighting Mechanisms for Improving Jacobian Estimates in the Adaptive Simultaneous Perturbation Algorithm'. In: *IEEE Transactions on Automatic Control* 54.6 (June 2009), pp. 1216–1229. ISSN: 0018-9286. DOI: `10.1109/TAC.2009.2019793` (cit. on p. 44).

[26] J.C. Spall. *Introduction to stochastic search and optimization: estimation, simulation, and control*. Wiley-Interscience series in discrete mathematics and optimization. Wiley-Interscience, 2003. ISBN: 9780471330523. URL: `http://books.google.com/books?id=3gsh1uJcRJEC` (cit. on p. 44).

[27] Malgorzata E. Wilinska et al. 'Simulation Environment to Evaluate Closed-Loop Insulin Delivery Systems in Type 1 Diabetes'. In: *Journal of Diabetes Science and Technology* 4.1 (Jan. 2010). PMID: 20167177 PMCID: 2825634, pp. 132–144. ISSN: 1932-2968 (cit. on p. 31).