Mineralization of terbuthylazine

Anders Nielsen

July 31, 2012

1 The problem

Terbuthylazine is a herbicide used in agriculture. It is a so-called s-triazin like atrazine, which has been banned in Denmark after suspicion of causing cancer. Terbuthylazine can be bound to the soil, but free terbuthylazine can be washed into the drinking water. Some bacteria can mineralize it. This data is part of a larger experiment to determine the ability of certain bacteria to mineralize terbuthylazine, and to estimate the mineralization rate.

1.1 System

The experiment starts out with all the terbuthylazine being free $F_0 = 100\%$, $B_0 = 0\%$, and $M_0 = 0\%$. Over the time of the experiment the terbuthylazine will gradually be bound to the soil (B), gradually be mineralized (M), and some of what is bound to the soil is released to be free (B). The system will be described via a system of ordinary differential equations:

$$\begin{aligned}
\frac{dB_t}{dt} &= -k_1 B_t + k_2 F_t, & B_0 &= 0 \\
\frac{dF_t}{dt} &= k_1 B_t - (k_2 + k_3) F_t, & F_0 &= 100 \\
\frac{dM_t}{dt} &= k_3 F_t, & M_0 &= 0
\end{aligned}$$

The system is closed, so the amount mineralized at any given time is

$$M_t = 100 - B_t - F_t,$$

so the system can be simplified by defining $X_t = (B_t, F_t)'$. The simplified system is:

$$\frac{dX_t}{dt} = \underbrace{\begin{pmatrix} -k_1 & k_2 \\ k_1 & -(k_2 + k_3) \end{pmatrix}}_{A} X_t, \qquad X_0 = \begin{pmatrix} 0 \\ 100 \end{pmatrix}$$

This system is a linear ODE system, so it can be solved. Here we will express the solution via the matrix exponential function.

$$X_t = e^{At} X_0$$

The amount mineralized is measured 26 times throughout a year.

A simple statistical model for these observations assumes independent normal measurement noise.

$$M_{t_i} \sim \mathcal{N}(100 - \Sigma X_{t_i}, \sigma^2), \quad \text{independent, and with } X_{t_i} = e^{At_i} X_0.$$

2 Implementation in AD Model Builder

2.1 Data

The data file prepared for reading into AD Model Builder is

```
# noObs
26
#  time terb
  0.77 1.396
  1.69 3.784
  2.69 5.948
  3.67 7.717
  4.69 9.077
  5.71 10.100
  7.94 11.263
  9.67 11.856
 11.77 12.251
 17.77 12.699
 23.77 12.869
 32.77 13.048
 40.73 13.222
 47.75 13.347
 54.90 13.507
 62.81 13.628
 72.88 13.804
 98.77 14.087
125.92 14.185
160.19 14.351
191.15 14.458
223.78 14.756
287.70 15.262
340.01 15.703
340.95 15.703
342.01 15.703
```

Table 1: Percentages mineralized at different times.

Time	**Mineralized**
0.77	1.396
1.69	3.784
2.69	5.948
3.67	7.717
4.69	9.077
5.71	10.100
7.94	11.263
9.67	11.856
11.77	12.251
17.77	12.699
23.77	12.869
32.77	13.048
40.73	13.222
47.75	13.347
54.90	13.507
62.81	13.628
72.88	13.804
98.77	14.087
125.92	14.185
160.19	14.351
191.15	14.458
223.78	14.756
287.70	15.262
340.01	15.703
340.95	15.703
342.01	15.703

2.2 AD Model Builder Code

The implementation follows the typical AD Model Builder template; first data is read in, then model parameters are declared, and finally the negative log likelihood is coded.

```
DATA_SECTION
  init_int noObs
  init_matrix obs(1,noObs,1,2)
  vector X0(1,2)

PARAMETER_SECTION
  init_vector logK(1,3);
  init_number logSigma;

  sdreport_vector k(1,3);
  sdreport_number sigma2;
  sdreport_vector M(1,noObs);

  matrix X(1,noObs,1,2);
  matrix A(1,2,1,2);
  objective_function_value nll;

PRELIMINARY_CALCS_SECTION
  X0(1)=0.0; X0(2)=100.0;
  logK=-2.0;
  logSigma=-2.0;

PROCEDURE_SECTION
  k=exp(logK);
  sigma2=exp(2.0*logSigma);

  A(1,1)= -k(1); A(1,2)=  k(2);
  A(2,1)=  k(1); A(2,2)= -k(2)-k(3);

  for(int i=1; i<=noObs; ++i){
    X(i)=expm(A*obs(i,1))*X0;
    M(i)=100.0-sum(X(i));
    nll+=0.5*(log(2.0*M_PI*sigma2)+square((obs(i,2)-M(i)))/sigma2);
  }
```

2.3 Running

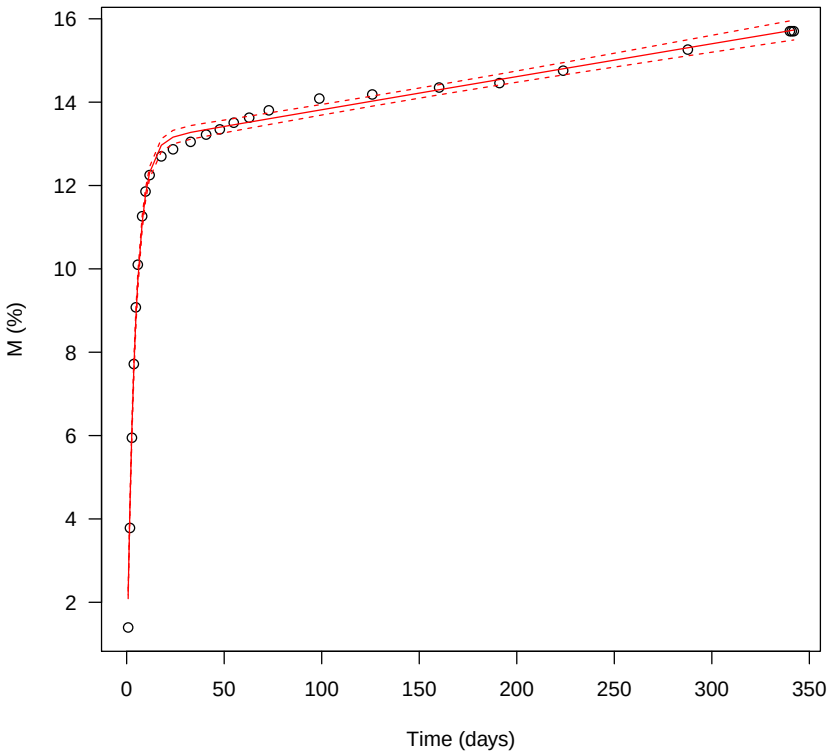
The model can be run from the command line by compiling and then executing the produced binary, but this can also be accomplished from within the R console like this:

```
> system('admb min')
> system.time(system('./min'))
```

as long as we ensure that the R working directory is set to the directory containing the AD Model Builder code for the problem `min.tpl` and the corresponding data file `min.dat`. The time to optimize the likelihood and calculate the Hessian was 0.613 seconds.

2.4 Results

After running the model we can plot the fit to make sure all went well.



Estimates and standard deviations of model parameters and derived quantities are located in the file `min.std`. From the figure it is evident that a more suitable model for this data set would include serial correlation, but for the purpose of comparing the three statistical tools with a simple we will ignore this for now.

3 Implementation in R

Using R Under development (unstable) (2012-07-27 r60013) and package versions:

```
Matrix    optimx    R2jags
 1.0-6 2012.5.24   0.03-07
```

3.1 Data

The same data file is used as for the AD Model Builder implementation, which is read in ignoring the first three lines in the file with the command:

```
> dat<-read.table('min.dat', skip=3, header=FALSE)
```

3.2 R code

The first attempt used the "optim" function in R with its default settings. The optimization was initialized by the same values as the AD Model Builder program, and parametrized in exactly the same way.

```
> dat<-read.table("../DATA/min.dat", skip=3, header=FALSE)
> library(Matrix)

> nlogL<-function(theta){
+     k<-exp(theta[1:3])
+     sigma<-exp(theta[4])
+     A<-rbind(
+              c(-k[1], k[2]),
+              c( k[1], -(k[2]+k[3]))
+              )
+     x0<-c(0,100)
+     sol<-function(t)100-sum(expm(A*t)%*%x0)
+     pred <- sapply(dat[,1],sol)
+     -sum(dnorm(dat[,2],mean=pred,sd=sigma, log=TRUE))
+ }
> (s1 <- system.time(fit<-optim(c(-2,-2,-2,-2),nlogL,hessian=TRUE)))
> fit$value
> fit$convergence
```

The default use of `optim` used 18 seconds, and reported successful completion. This is however not the correct solution. The AD Model Builder program reported a minimum negative log likelihood of 0.939, but R reported a minimum of 19.269. Of the built-in optimizers in optim only "L-BFGS-B" managed to find the correct solution.

```
> system.time(fit<-optim(c(-2,-2,-2,-2),nlogL,hessian=TRUE, method='L-BFGS-B'))
> fit$value
> fit$convergence
```

The recent add-on package to R called "optimx" allows the user to try out many different build-in optimizers for a given objective function, and compare their solutions, and running time.

```
> library(optimx)

> fit <- optimx(c(-2,-2,-2,-2),nlogL,hessian=TRUE,control=list(all.methods=TRUE))

   method                                          par   fvalues  fns grs hes
13 bobyqa       8.314150, -11.028827, 4.638480, 4.477432  153.3056  157   0   0
9  Rcgmin    -19.185319, 17.416518, -44.176230, 2.533604  102.7661   85  50   0
```

```
5     nlm        -24.57290, 23.50660, -57.43591, 2.53360   102.766  182   0   0
10 Rvmmin    -27.551412, 26.872030, -64.750702, 2.533601   102.766   73  14   0
2    BFGS    -27.551412, 26.888313, -64.750702, 2.533601   102.766   89  16   0
3      CG    -14.052437, 11.626626, -31.598186, 2.533602   102.766  616 100   0
1      NM -7.2485767, -1.4200733, -3.3243726, -0.3518798  19.26905  225   0   0
15   nmkb     -7.249600, -1.582477, -3.476082, -1.382858 0.9392151  357   0   0
6  nlminb     -7.249619, -1.582472, -3.476072, -1.382815 0.9392142   43  30   0
8  ucminf     -7.249616, -1.582472, -3.476072, -1.382814 0.9392142   65  57   0
4  LBFGSB     -7.249617, -1.582471, -3.476071, -1.382819 0.9392142   83  75   0
7     spg     -7.249619, -1.582466, -3.476066, -1.382815 0.9392142  267 182   0
14   hjkb     -7.249611, -1.582466, -3.476067, -1.382816 0.9392142 1222   0   0
12 newuoa     -7.249611, -1.582464, -3.476065, -1.382815 0.9392142 1906   0   0
11 uobyqa     -7.249611, -1.582464, -3.476065, -1.382815 0.9392142  479   0   0
   rs conv  KKT1  KKT2       mtilt xtimes  meths
13  0    0  TRUE FALSE 0.007277456 10.457 bobyqa
9   0    3  TRUE FALSE          NA 21.873 Rcgmin
5   0    3  TRUE FALSE          NA 11.569    nlm
10  0    3  TRUE FALSE          NA  9.724 Rvmmin
2   0    0  TRUE FALSE 0.002090308 10.829   BFGS
3   0    1  TRUE FALSE 0.001548984 73.568     CG
1   0    3 FALSE FALSE          NA 15.325     NM
15  0    0 FALSE  TRUE    9.817502 23.013   nmkb
6   0    0  TRUE  TRUE   0.9040154 11.849 nlminb
8   0    0 FALSE  TRUE    1.150713 21.505 ucminf
4   0    0 FALSE  TRUE    1.024187 30.826 LBFGSB
7   0    0 FALSE  TRUE   0.6465689 78.049    spg
14  0    0 FALSE  TRUE   0.9004727 82.205   hjkb
12  0    0 FALSE  TRUE    0.704133 125.48 newuoa
11  0    0 FALSE  TRUE   0.7107458 31.094 uobyqa
```

From this table we see that for this particular model the fastest optimizer in R which actually found the correct minimum was nlminb and the time was ca. 12 seconds. Of the 15 built-in optimizers, 8 did find the correct solution. Based on the new convergence criteria (the columns `KKT1` and `KKT2` where both should be TRUE), only the `nlminb` model fit is unproblematic.

4 Implementation in BUGS (jags)

4.1 Data

The same data file is used as for the AD Model Builder implementation.

4.2 BUGS code

The implementation in BUGS comes in two parts; (1) the BUGS model description (in a separate file), and (2) an R script reading in data and calling the model with data and specifying the starting

points for the chains and desired number of iterations. First the model description file:

```
model {
  A[1,1] <- -exp(logK1)
  A[1,2] <-  exp(logK2)
  A[2,1] <-  exp(logK1)
  A[2,2] <- -exp(logK2)-exp(logK3)

  tau <- 1/(sigma*sigma)
  x0[1] <- 0
  x0[2] <- 100
  for (i in 1:noObs){
    M[i] ~ dnorm (pred[i], tau)
    pred[i] <- 100-sum(mexp(A*time[i])%*%x0)
  }
  sigma ~ dunif (0, 1000)
  logK1 ~ dunif(-10,10)
  logK2 ~ dunif(-10,10)
  logK3 ~ dunif(-10,10)
}
```

The next section will show the second part.

4.3 Running

The R commands for reading in data and parsing it to the BUGS model.

```
> library(R2jags)
> load.module('msm') # the module containing the matrix exponential

> dat <- read.table('../DATA/min.dat', skip=3, header=FALSE)
> time <- dat[,1]
> M <- dat[,2]
> noObs <- length(time)
> set.seed(12345)
> jags.data <- list("noObs","time","M")
> jags.params <- c("logK1","logK2","logK3","sigma")
> jags.inits <- function(){
+   list("logK1"=-2+.2*rnorm(1),"logK2"=-2+.2*rnorm(1),"logK3"=-2+.2*rnorm(1),
+         "sigma"=exp(-2+.2*rnorm(1)), .RNG.name="base::Wichmann-Hill", .RNG.seed=round(runif(1
+ }

> time<-unname(system.time(
+    jagsfit <- jags(data=jags.data, inits=jags.inits, jags.params,
+                    n.iter=10000, n.thin=1,n.burnin=0,model.file="model.txt")
+  )["elapsed"])
```

Running the 3 chains with 10000 iterations in each took approximately 51 seconds.

4.4 Results