PNNL-14051

Pacific Northwest
National Laboratory
Operated by Battelle for the
U.S. Department of Energy

S-PLUS Library For Nonlinear Bayesian Regression Analysis

P.G. Heasler
K.K. Anderson
J.L. Hylden

September 2002

Prepared for the U.S. Department of Energy
under Contract DE-AC06-76RL01830

PACIFIC NORTHWEST NATIONAL LABORATORY
operated by
BATTELLE
for the
UNITED STATES DEPARTMENT OF ENERGY
under Contract DE-AC06-76RL01830

PNNL-14051

S-PLUS Library For Nonlinear Bayesian Regression Analysis

P.G. Heasler
K.K. Anderson
J.L. Hylden

September 2002

Prepared for
the U.S. Department of Energy
under Contract DE-AC06-76RL01830

Pacific Northwest National Laboratory
Richland, Washington 99352

Contents

1 Problem Description

This report describes a library of S-PLUS functions that have been constructed to support two LDRD's. These LDRD's are (1) "Bayesian Nonlinear Regression Modeling for Remote Sensing Applications" supported under the CSE initiative, and (2) "Development of an Improved Clutter Suppression Model for Plume Detection," supported under the Infrared Sensing Initiative. S-PLUS is distributed by Insightful Corporation, Seattle, WA.

This library constitutes our prototype algorithms for non-linear Bayesian regression and IR estimation. We expect these functions to be improved and revised, but hopefully this library is mature enough to be used without major modification for the rest of the year.

This S-PLUS library has been developed to solve a certain class of Nonlinear Bayesian Regression Problems. The basic problem is described by the regression model;

$$Y = F(\beta, data) + E \tag{1}$$

where Y represents a vector of measurements, and $F(\beta, data)$ represents a S-PLUS function that has been constructed to describe the measurements. As one can see, $F(\beta, data)$ depends upon β, a vector of parameters to be estimated, while $data$ is an S-PLUS object containing any other information needed by $F()$. The errors, E, are assumed to be independent, normal, unbiased and to have *known* standard deviations of $stdev(E) = sd.E$.

A prior is imposed on β and is represented by $p(\beta)$. The family of prior functions that we have currently chosen for this problem is an *Extended Beta* family. This is a beta distribution that is defined over an arbitrary interval instead of (0,1). Each component of β is assumed to be independent, so the prior is the product of univariate extended betas. This means that the problem has to be formulated in terms of parameters that are independent with respect to the prior information.

In our IR modeling application, the components in β can generally be split into two groups; estimation parameters and nuisance parameters. The prior on the estimation parameters will generally be non-informative, while those on the nuisance parameters will be constructed to reflect the information we have about them. We hope the extended beta distribution is general enough to adequately represent the information we have on the nuisance parameters.

The estimation algorithms have been written so that it would be very easy to substitute another family of prior's for the extended beta family. However any substituted family must be (1) independent and (2) non-zero within a defined hyper-cube. For example, a product of truncated normals would satisfy this requirement. It should also be noted that some (or all) of the hyper-cube boundaries can be infinite.

1.1 Estimators Desired

Two estimation solutions are desired for the described regression problem; (1) We want to calculate the *Maximum Posterior Density* (MPD) estimate for the problem (with the associated covariance matrix), and (2) the marginal posterior distribution for each component

in β. The posterior distribution, $f(\beta|Y)$, is easily written down. It is;

$$f(\beta|Y) = C_0 \exp\left(-\frac{1}{2}\sum_i \frac{(Y_i - F_i(\beta, data))^2}{sd.E_i^2}\right) p(\beta) \tag{2}$$

The maximum posterior density is defined as the value of β that maximizes this density. It can be calculated with an iterative minimization technique that closely resembles a classical non-linear regression algorithm. For this problem, the "sum of squared error" function is defined by;

$$SSE(\beta) = \sum_i \frac{(Y_i - F_i(\beta, data))^2}{sd.E_i^2} - 2\log(p(\beta)) \tag{3}$$

and the SSE is minimized to find $\hat{\beta}$. This sort of algorithm produces an estimate relatively quickly and was developed to produce an improved clutter suppression model of the Infrared Sensing Initiative LDRD. As one would expect, maximum posterior density estimation requires as much computer time as classical non-linear regression. MPD estimation works well when the problem is (1) approximately linear and (2) the priors are approximately normal.

Estimation of the Bayesian posterior distribution can only be accomplished through Monte Carlo techniques (i.e., sampling from the posterior distribution). This is accomplished with two Markov Chain Monte Carlo (MCMC) algorithms, the Metropolis-Hastings and the Hamiltonian method. These techniques require much more computational time than MPD estimation. An MPD estimate may require 50 evaluations of the function, $F(\beta, data)$, while a Markov Chain algorithm typically requires 25,000 or more evaluations. Hence, MCMC estimation is much more computationally intensive than HPD estimation. Even though MCMC estimation is slow, it produces a correct posterior distribution for the β as contrasted to an asymptotic approximation. Consequently the MCMC posterior is much more acceptable when the problem is highly non-linear and the priors non-normal.

A MCMC estimator actually allows us to calculate any integral of the form;

$$\int H(\beta) f(\beta|Y)\, d\beta \tag{4}$$

For example, if $H(\beta) = \beta_k$, we obtain the mean of β_k. Our focus will be to use the MCMC to compute the following statistics for β; the mean, standard deviation, mode, minimum, maximum, and selected quantiles.

1.2 Problem Variants and Extentions

Here is a list of some variations on the basic problem that we would also like to be able to solve, either by extending the functionality of our basic algorithms, or with the construction of separate algorithms.

1. Use of an arbitrary (user-supplied) continuous prior in the algorithms.
2. Assume that Var(E) is an unknown parameter and must be estimated from the data.
3. Use of discrete priors on some components in β.

2 Estimation Algorithm Overview

The estimation algorithms have been constructed so that their inputs, outputs, and operation are as similar as possible. This section describes certain common operating features and data objects used by all the estimation algorithms.

2.1 Common Operating Features

Fixed Variables: A subset of components in the parameter vector, β, can be "fixed," which means that they are set at specific values. The algorithms then calculate the estimates, conditional on the fixed values. Logically, this is equivalent to specifying degenerate distributions for those fixed components. Consequently, fixed parameters are specified by setting the prior standard deviation of that parameter to zero, and the prior mean to the desired fixed value.

Priors defined over a Hypercube: The user defined prior, $p(\beta)$ is assumed to be positive within a user-defined hypercube. The bound defining the hypercube may be finite or infinite.

Posterior Distribution Statistics: The MCMC estimators produce the following posterior statistics on each component in the β-vector: mean, mode, standard deviation, minimum, maximum.

Output of a M.C. Sample: The MCMC estimators produce a "sample" of a user specified subset of components in the β-vector. The output may be "thinned" to one sample in N, with N being specified by the user. Thinning is used to reduce storage requirements.

2.2 Common Data Objects

We want to create a set of algorithms that can be applied to the same regression problem with the minimum of data reformatting. To make this easy to do in S-PLUS, a common argument list is used in all of the estimation algorithms. The data objects in this argument list are defined below;

Fo(beta,Fdata,der=T): This is a user supplied function that defines the regression model. *beta* contains the regression model parameters, and $Fdata$ is any other information that Fo needs in its calculation. Fo must produce a list with two components, Fo and dF. Fo is a vector of $length(Y)$ containing the function evaluation. The other component, dF is the derivative of the function with respect to *beta*. Consequently, it is a matrix of $length(Y)$ by $length(beta)$. If $der = T$, the derivative is calculated and stored in dF, otherwise, when $der = F$, $NULL$ is stored in dF.

lg.prior(beta,pdata,der=T): This is a user supplied function that defines the prior distribution. It calculates $log(p(\beta_i))$ for the parameter vector *beta*. Information necessary to define the prior distribution is contained in *pdata*, and can contain anything the

user may choose to include. There are some restrictions on the prior, however. The prior is assumed to be zero outside of a hypercube, which is defined in pdata$stats.

The function returns a list with components *lpr* and *dlpr*. *lpr* contains the log-prior associated with *beta*, while *dlpr* contains the derivative with respect to beta. If der=F, *dlpr* is set to NULL and not calculated, while der=T causes *dlpr* to be calculated.

lpr can be set to $-Inf$, which indicates that the prior is zero at that point. When $stats[i, 2] = 0$ (i.e., standard deviation is zero), component i in *beta* is considered fixed, and both *lpr* and *dlpr* should be set to NA.

Fdata: This is a list to be passed to a regression function, Fo. It has whatever data is required by Fo. However, it must also include the following components, which are used by the estimation algorithms;

Y: The data vector for the algorithm.

sd.E: The standard deviations of the terms in the error vector E.

pdata: This is a list containing the information required to calculate the prior density; It will have any components required by *lg.prior* but must also contain the component *stats*, which is used by the estimation algorithms. *stats* is an $length(beta)$ by 4 table that contains the mean, stdev, min and max of each component in *beta*. This table contains the distribution parameters for the extended beta.

The second column, stdev, tells whether or not a parameter is fixed. If the stdev is set to zero, the parameter is fixed at the mean. The min and max can be set to $-Inf$ and Inf for unbounded distributions.

start: The starting value for the estimation algorithm. The default is the prior mean from $pdata\$stats[, 1]$.

control: This is a list containing parameters that control the operation of the algorithm. The format of control varies from algorithm to algorithm.

3 Maximum Posterior Density Algorithms

This set of algorithms calculates the maximum posterior density estimate for the regression problem plus an asymptotic covariance matrix. One algorithm, *marginal.post* calculates an experimental approximation the marginal posterior distribution of each component in β. This experimental approximation is much quicker than the Monte Carlo techniques presented in the next section.

3.1 nlmpd

3.1.1 Function Description

This is a general non-linear maximum posterior density estimator. The calling sequence is:

```
nlmpd(Fo,Fdata,lg.prior,pdata,start=NULL,control=NULL);
```

Input: See Section 2.2 for a description of the arguments *Fo*, *Fdata*, *lg.prior* and *pdata*. It should be noted that $pdata\$stats[,2]$ defines which parameters are fixed or free; Those that are fixed have their standard deviations set to zero in $stats[,2]$.

The *control* argument is a list of parameters that controls algorithm performance. See the Algorithm Operation section for an exact description of what these parameters do. Here are the components in control, with default settings;

marq.par: The starting value for α in the algorithm (default $marq.par = 1$).

marq.scale: The value for λ in the algorithm (default $marq.scale = 2$).

max.iter: The maximum number of iterations allowed by the algorithm (default $max.iter = 50$).

grad.tol: The convergence tolerance for the gradient vector (default $grad.tol = 1e{-}4$).

max.step: The maximum number of Marquardt steps allowed (default $max.step = 10$).

sse.tol: The convergence tolerance for the sum of squared error, $-2log(posterior)$ (default is $sse.tol = 1e-7$).

Output: *nlmpd* returns a list with the following components;

beta: MPD estimate for the problem.

sd.beta: This contains the standard deviations of the beta, both fixed and free.

cov: This contains the asymptotic covariance matrix of only the free beta.

sse: The sse associated with the beta estimate (with $Nobs + Nfp$ subtracted out).

Nfp: The number of free parameters in the model.

Nobs: The number of observations, $length(Y)$.

diag: A list containing algorithm diagnostics. It contains *beta*, which is a matrix of beta vectors, one for each iteration, and *it.par*, a table containing the marquardt parameter, sse, gradient tolerance, and sse tolerance for each iteration.

control: This contains the control parameters used by the algorithm.

Function Status: Several variants of this algorithm were built and are stored in the directory "nlmpd.lib." The variants have different strategies for dealing with reflection on the distribution boundary. None of the variants works 100% of the time. The installed variant uses a version of the Marquardt-Levenburg algorithm. We also tried to use "nlregb" which should automatically solve the problem.

This algorithm can get stuck at a non-minimum on the boundary.

3.1.2 Algorithm Operation

This description will ignore the fixed parameters in β, so β will only represent free parameters. Let $SSE(\beta)$ represent the weighted sum of squared error;

$$SSE(\beta) = \sum_i \left[\frac{Y_i - Fo(\beta, data)_i}{sd.E_i}\right]^2 - 2\sum_j lg.prior(\beta, pdata)\$lpr_j \tag{5}$$

Minimization of this function produces the MPD estimate. $SSE(\beta)$ is actually the log-posterior multiplied by -2. We minimize the function by applying a Marquardt-Levenburg algorithm. At step $i+1$, we produce an estimate β_{i+1} from β_i by solving the following linear equation;

$$-G_i = (H_i + (1+\alpha)D_i)\delta\beta \tag{6}$$

In the above equation, G_i is the gradient vector of $SSE(\beta_i)$, H_i and D_i are matricies defined below, and $\delta\beta$ is to be solved for. The new estimate, $\beta_{i+1} = \beta_i + \delta\beta$. It so happens that the new beta may not produce a smaller SSE, unless α is set large enough. So if $SSE(\beta_{i+1}) > SSE(\beta_i)$, α is increased by the factor λ until the SSE is smaller. If the SSE is smaller, then alpha is decreased by the factor λ for use in the next iteration.

The gradient, G_i is defined by;

$$-G_i = \sum_j \frac{Y_j - Fo_j}{sd.E^2} dFo_j + \sum_j dlpr_j \tag{7}$$

and the matricies are defined as;

$$H_i = \sum_j \frac{dF_j dF_j^T}{sd.E^2} \tag{8}$$

and

$$D_i = stats[i,2]^{-2} \tag{9}$$

Iteration continues until one of the following criteria is met;

1. Number of iterations exceeds $max.iter$.

2. $|G_i * prior * sd| < Nfp * grad.tol$
3. $|SSE_i - SSE_{i-1}| < Nfp * sse.tol$

One final issue needs to be discussed, and that is, what to do when a prospective point falls outside of the hypercube of positive probability, defined by *stats*. In this case, adjust the Marquardt parameter, α so that the new point is "just inside" the hypercube and continue operation.

3.1.3 S-PLUS Code

```
nlmpd <- function(Fo, Fdata, lg.prior, pdata,
                  start = NULL, control = NULL, debug = F) {
# ARGUMENTS:
#  Fo: Splus function describing the model.
#  Fdata: A list passed to Fo containing any misc data Fo
#        requires AND data Y, stdev(E)=sd.E.
#  pdata: a list required by lg.prior.  Contains the table "stats" which has
#         means, standard deviations, mins and maxs of the prior distr.
#  start: starting values for beta. Default is mean from "stats"
#         starting values must be in the interior of the hypercube.
#  control: a vector of length 5 that controls operation of the
#        minimization algorithm. it contains: Marquard parameter,
#        step inflator, max # of iterations, relative gradient
#        tolerance, and maximum number of inflation steps.
#  debug=F; controls printing out results.
# VALUE: Result is a list containing:
#    beta: estimate of parameters.
#    sd.beta: standard deviation on parameters.
#    cov: Asymptotic covariance matrix of estimate.
#    fitted: Yhat for model
#    diag: a list containing diagnostics for algorithm.
#       beta: beta value for each iteration
#       it.par: control parameters for each iteration.
#    control
stats <- pdata$stats
if(is.null(start)) start <- stats[, 1]
Y <- Fdata$Y
sd.E <- Fdata$sd.E
Np <- length(start)
Nobs <- length(Y)
psd <- stats[, 2]
fr1 <- psd > 0
D <- 1/psd[fr1]^2
We <- 1/sd.E^2
```

```
Nfp <- sum(fr1)
if(is.null(control)) {
control <- c(1, 2, 50, 1.0e-05, 10,
1.0e-07, 0.001)
}
names(control) <- c("marq.par", "marq.scale", "max.iter", "grad.tol",
"max.step", "sse.tol", "boundary.tol")
maxit <- control[3]
# calculate boundaries;
wt <- control[7]
lower <- stats[, 3]
upper <- stats[, 4]
u <- lower !=  - Inf
lower[u] <- (1 - wt) * lower[u] + wt * stats[u, 1]
u <- upper != Inf
upper[u] <- (1 - wt) * upper[u] + wt * stats[u, 1]
# make up output arrays
beta <- array(0, dim = c(Np, maxit))
cres <- array(0, dim = c(5, maxit))
dimnames(cres) <- list(c("marq.par", "SSE", "grad.tol",
                         "sse.tol","Nfp"),
NULL)
beta[, 1] <- pmax(pmin(start, upper), lower)
Fold <- Fo(beta[, 1], Fdata, der = T)
Fold$dF <- Fold$dF[, fr1]
pres <- lg.prior(beta[, 1], pdata, der = T)
SSE <- sum(We * (Y - Fold$Fo)^2 - 1) - 2 *
sum(pres$lpr[fr1])
Gm <- as.vector((We * (Y - Fold$Fo)) %*% Fold$dF) +
pres$dlpr[fr1]
H <- t(Fold$dF * We) %*% Fold$dF
DH <- diag(H)
fr2 <- fr1 & beta[,1]>lower & beta[,1]<upper;
tol.grad <- max(abs(Gm[fr2[fr1]] * stats[fr2, 2]))
tol.sse <- abs(SSE)/Nfp
cres[, 1] <- c(control[1], SSE, tol.grad, tol.sse,sum(fr2))
i <- 2
alpha <- cres[1, 1]
maxj <- control[5]
Irate <- T

while(Irate == T) {
  # find correct step size....
```

```
  j <- 0
  while(SSE >= cres[2, i - 1] & j < maxj) {
j <- j + 1
diag(H) <- DH + (1 + alpha) * D
fr4<-fr2[fr1];
dB <- chol.solve(H[fr4,fr4], Gm[fr4])
beta[!fr2,i] <- beta[!fr2,i-1];
beta[fr2, i] <- beta[fr2,i-1] + dB;
# Make sure new estimate is in hypercube
fr3 <- fr2 & beta[, i] > lower & beta[, i] < upper
if(!all(fr3 == fr2)) {
  # naive fix of problem...
  zl<-lower[fr2]!=beta[fr2,i-1];
  zu<- upper[fr2]!=beta[fr2,i-1];
  scl <- max(c((dB/(lower[fr2] - beta[fr2, i - 1]))[zl],
          (dB/(upper[fr2] - beta[fr2, i - 1]))[zu], 1));
  beta[fr2, i] <- beta[fr2, i - 1] + dB
  beta[, i] <- pmin(pmax(lower,beta[, i]), upper);
  # cat("One");
  # browser();
  fr2 <- fr2 & beta[, i] > lower &
beta[, i] < upper;
}
Fold <- Fo(beta[, i], data = Fdata, der = T)
Fold$dF <- Fold$dF[, fr1]
pres <- lg.prior(beta[, i], pdata, der = T)
SSE <- sum(We * (Y - Fold$Fo)^2 - 1) - 2 * sum(pres$lpr[fr1])
if(SSE > cres[2, i - 1]) {
alpha = alpha * control[2]
} else {
alpha = alpha/control[2]
}
} # inner while
  Gm[] <- as.vector((We * (Y - Fold$Fo)) %*% Fold$dF) +
           pres$dlpr[fr1];
  fix <- fr1 & !fr2
  if(any(fix)) {
    flow<-beta[,i]==lower;
    fr2[fr1] <- fr2[fr1] | (flow[fr1] & Gm>0);
    fhi<-beta[,i]==upper;
    fr2[fr1] <- fr2[fr1] | (fhi[fr1] & Gm<0);
    # cat("two");
    # browser();
```

```
  }
  H[] <- t(Fold$dF * We) %*% Fold$dF
  DH[] <- diag(H)
  tol.grad <- max(abs(Gm[fr2[fr1]] * stats[fr2, 2]))
  tol.sse <- abs(SSE - cres[2, i - 1])/Nfp
  cres[, i] <- c(alpha, SSE, tol.grad, tol.sse,sum(fr2));
  if(debug) {
   cat("Done with iteration", i, "\n")
   print(cres[, i])
  }
  Irate<-(tol.grad>control[4] | tol.sse>control[6]) & i < maxit;
  if(Irate == T) i <- i + 1;
} # end of while
bname <- dimnames(stats)[[1]]
beta <- beta[, 1:i]
cres <- cres[, 1:i]
dimnames(cres) <- list(c("marq.par", "SSE", "grad.tol",
                         "sse.tol","Nfp"), NULL);
dimnames(beta) <- list(bname, NULL)
diag(H) <- DH + D
cov.b <- chol.solve(H, diag(rep(1, sum(fr1))))
bname <- dimnames(stats)[[1]]
dimnames(cov.b) <- list(bname[fr1], bname[fr1])
sdbeta <- numeric(Np)
sdbeta[fr1] <- sqrt(diag(cov.b))
return(list(beta = beta[, i], sd.beta = sdbeta, cov = cov.b,
    fitted =Fold$Fo, sse = cres[2, i], Nfp = Nfp, Nobs = Nobs,
    fr2=fr2,
    diag = list( beta = beta, it.par = cres), control = control))
} # End Function...

# This function can get stuck in the following manner;  If many
# parameters are fixed, and some Gm values indicate some should be
# made free.  The next iteration could result in a dB in which one
# of them should again be fixed.  However, this will make scl
# infinite, and beta[,i]=beta[,i-1] for all future iterations.
# I put in a naive fix of problem...
```

3.2 nlmpd.norm

This is a non-linear maximum posterior density estimator that uses a normal distribution for the prior.

```
nlmpd.norm(Fo,Fdata,pdata,start,control)
```

3.2.1 Function Description

Input: Arguments are defined the same as for the function *nlmpd*. *pdata*$*stats* contains the means and standard deviations of the normal priors. The *min* and *max* columns are not used in this algorithm.

Output: The output is the same as *nlmpd*.

Function Status: This function was originally called *nlmap*.

3.2.2 Algorithm Operation

The algorithm operation is the same as *nlmpd*, except that it automatically builds the *log.prior* distribution and does not need to consider boundaries.

3.2.3 S-PLUS Code

```
nlmpd.norm <- function(Fo,Fdata,pdata,start=prior[,1],
           free=rep(T,dim(pdata)[1]),control=NULL,debug=F){

# DESCRIPTION: Calculates the MAP (maximum apost. probability)
#       for a Bayesian regression problem, assuming a normal
#       prior distribution.
# ARGUMENTS:
#  Fo: Splus function describing the model.
#  Fdata: An object passed to Fo containing any misc data Fo
#        requires AND data Y, stdev(E)=sd.E.
#  pdata: a table containing means and standard deviations
#         of the prior distr. of beta. mean(beta)=pdata[,1],
#         and stdev(beta)=pdata[,2].
#  start: starting values for beta.
#  free: parameters in Fo that are considered free.
#        Others are fixed at starting values.
#  control: a vector of length 5 that controls operation of the
#        minimization algorithm. it contains: Marquard parameter,
#        step inflator, max # of iterations, relative gradient
#        tolerance, and maximum number of inflation steps.
#  debug=F; controls printing out results.
# VALUE: Result is a list containing:
```

```
#    beta: estimate of parameters.
#    sd.beta: standard deviation on parameters.
#    cov: Asymptotic covariance matrix of estimate.
#    fitted: Yhat for model
#    diag: a list containing diagnostics for algorithm.
#       beta: beta value for each iteration
#       it.par: control parameters for each iteration.
#    control
# NOTE: This has been revised to (1) include a control parameter
#    on delta-SSE, and (2) to allow one to automatically
#    "fix" parameters, (3) more complete output.

Y<-Fdata$Y;
sd.E<-Fdata$sd.E;

Np<-length(start);
Nobs<-length(Y);

Eb<-pdata[,1];
Wb<-1/pdata[,2]^2;
We<-1/sd.E^2;
Nfp<-sum(free);

if(is.null(control)){
control<-c(1,2,50,1.0e-5,10,1.0e-7)
}
names(control)<-c("marq.par","marq.scale","max.iter",
                  "grad.tol","max.step","sse.tol");
maxit<-control[3];

# make up output arrays
beta<-array(start,dim=c(Np,maxit));
cres<-array(0,dim=c(4,maxit));
# grad<-beta;
dimnames(cres)<-list(c("marq.par","SSE","grad.tol","sse.tol"),
                     NULL);
beta[,1]<-start;
Fold<-Fo(beta[,1],Fdata=Fdata);
Fold$dF<-Fold$dF[,free];

SSE<-sum(We*(Y-Fold$Fo)^2-1) +
     sum(Wb*(beta[,1]-Eb)^2-1);
XtY<- as.vector((We*(Y-Fold$Fo)) %*% Fold$dF) +
```

```
                (Eb[free]-beta[free,1])*Wb[free];

XtX<- t(Fold$dF*We) %*% Fold$dF;
H<-XtX;
tol.grad<- max(abs(XtY*pdata[free,2]))/Nfp
tol.sse<- abs(SSE)/Nfp
cres[,1]<-c(control[1],SSE,tol.grad,tol.sse);

i<-2;
alpha<-cres[1,1];
maxj<-control[5];

Irate<-T;
while(Irate==T){
# find correct step size....
j<-0;
while(SSE>=cres[2,i-1] & j<maxj){
  j<-j+1
  diag(H)<-diag(XtX)+(1+alpha)*Wb[free];
  dB<-chol.solve(H,XtY);
  beta[free,i]<-beta[free,i-1]+dB;
  Fold<-Fo(beta[,i],Fdata=Fdata);
  Fold$dF<-Fold$dF[,free];

  SSE<-sum(We*(Y-Fold$Fo)^2-1) +
       sum(Wb*(beta[,i]-Eb)^2 -1);
       if(SSE>cres[2,i-1]) {
           alpha=alpha*control[2];
       } else {
           alpha=alpha/control[2];
       }
} # inner while

XtY[]<- as.vector((We*(Y-Fold$Fo)) %*% Fold$dF) +
                (Eb[free]-beta[free,i])*Wb[free];
XtX[]<- t(Fold$dF*We) %*% Fold$dF;
H[]<-XtX;
tol.grad<- max(abs(XtY*pdata[free,2]))/Nfp;
tol.sse<- abs(SSE-cres[2,i-1])/Nfp;


cres[,i]<-c(alpha,SSE,tol.grad,tol.sse);
if(debug){
```

```
  cat("Done with iteration",i,"\n");
  print(cres[,i]);
}
Irate<-(tol.grad>control[4] & i<maxit &
                  tol.sse>control[6]) ;
if(Irate==T) i<-i+1;
} # end of while

bname<-dimnames(pdata)[[1]]
beta<-beta[,1:i];
cres<-cres[,1:i];
dimnames(cres)<-list(c("marq.par","SSE","grad.tol","sse.tol"),
                     NULL);
dimnames(beta)<-list(bname,NULL);

diag(H)<-diag(XtX)+Wb[free];
cov.b<-chol.solve(H,diag(rep(1,sum(free))));
bname<-dimnames(pdata)[[1]];
dimnames(cov.b)<-list(bname[free],bname[free]);

sdbeta<-numeric(Np);
sdbeta[free]<-sqrt(diag(cov.b));

return(list(beta=beta[,i],sd.beta=sdbeta,
            cov=cov.b,fitted=Fold$Fo,
            sse=cres[2,i],
            Nfp=Nfp,
            Nobs=Nobs,
            diag=list(beta=beta,it.par=cres),
            control=control));
} # End of Function....
```

3.3 marginal.post

This subroutine uses *nlmpd* to explore the log-posterior surface and create an approximation to the marginal posterior for designated components in the β-vector. This routine uses *nlmpd* to find the minimum, and then calculates minimum log-posterior points with one component in β fixed. These results are then used to construct an approximation to the conditional distribution for that component.

```
marginal.post(Fo,Fdata,lg.prior,pdata,marg.comp=NULL,control=NULL
              width=3,full.output=F)
```

3.3.1 Function Description

Input: See Section 2.2 for a description of the arguments, *Fo*, *lg.prior*, and *pdata*. Below is an explanation of those not described in that section;

marginal.comp: This vector identifies the marginal distributions to be computed. This vector contains the id's of the components in β. (default is all components in β).

control: This contains control information for *nlmpd*, see *nlmpd* for a description of *control*.

width: The size of the area to be investigated around the MPD estimate. To calculate the marginal, the algorithm will assume essentially all the probability is between $MPD \pm width * stderr$.

full.output: If false, only the marginal distributions are produced. If true, the MPD ridge for each component in *marginal.comp* is produced.

Output: Output is a list with the following components:

marg.comp: Same as input argument.

beta: MPD estimate for regression problem. If *full.output* is true, beta is a 100 by $length(marg.comp)$ by $length(\beta)$ that contains the MPD ridges for each designated component.

sd.beta: Asymptotic standard error for beta.

marg.cdf: This is a 100 by $length(marg.comp)$ by 2 array that contains the marginal distributions. $marg.cdf[,i,1]$ contains the β values for component $marg.comp[i]$, while $marg.cdf[,i,2]$ contains the associated cumulative distribution function values.

Function Status: This function was previously called *explore.map*.

3.3.2 Algorithm Operation

Let us consider the calculation of the marginal for component i in the parameter vector, β and to do this, re-write the vector β as (z, u), where z represents component β_i and u all other $(p-1)$ components of β. Also let $f(z, u)$ represent the posterior distribution of interest. We have;

$$f(z,u) = C_0 \exp(-\frac{1}{2} SSE(z,u)) \tag{10}$$

where C_0 is an integration constant and $SSE(z, u)$ is the sum of squared error for the regression problem (defined by Equation 3). *nlmpd* is used to find the MPD estimates for u at a set of fixed values of z; denote the MPD estimate by $\hat{u}(z)$ and the associated asymptotic covariance matrix by C_z.

We want to calculate points on the marginal distribution, $f(z)$, which is defined by;

$$f(z) = \int f(z,u)\, du \tag{11}$$

This is difficult, because a high-dimensional integral has to be evaluated. To get rid of the high-dimensional integral, the following approximation for the posterior is used;

$$f(z,u) \approx C_0 \exp(-\frac{1}{2}[SSE(z,\hat{u}(z)) + (u-\hat{u}(z))^T C_z^{-1}(u-\hat{u}(z))]) \tag{12}$$

So the approximation assumes that the components represented by u are approximately normal. If this approximation is integrated over u, we get;

$$\begin{aligned} f(z) &= \int f(z,u)\, du && (13)\\ &= C_0 \exp(-\frac{1}{2} SSE(z,\hat{u}(z)))(2\pi)^{(p-1)/2}\sqrt{det(C_z)} && (14)\\ &= C_1\sqrt{det(C_z)} \exp(-\frac{1}{2} SSE(z,\hat{u}(z))) && (15) \end{aligned}$$

To calculate the marginal distribution, $f(z)$, N values $(z_1, z_2 ... z_N)$, are uniformly spaced in the interval $\hat{\beta}_i \pm width * stderr(\hat{\beta}_i)$ and the above formula formula is used to calculate $f(z_k)$. The calculation must determine the unknown normalization constant, C_1, and this is done by making the discrete distribution integrate to 1.

3.3.3 S-PLUS Code

```
mariginal.post <- function (Fo, prior,data,free=rep(T,dim(prior)[1]),
            control=NULL,expl.par=free,cond.sse=T,expl.width=3){
  Nev<-20;
  Nep<-sum(expl.par);
  idb<-seq(expl.par)[expl.par];
  Y<-data$Y;
  sd.E<-data$sd.E;
```

```
  fit0<-nlmap(Fo,prior,data,free=free,control=control);
  fit<-fit0
  beta0<-fit$beta;
  sdb<-fit$sd.beta;
  beta<-matrix(0,length(beta0),1 + 2*Nev*Nep);
  SSE<-numeric(dim(beta)[2]);
  SSE[1]<-fit$sse;
  beta[,1]<-fit$beta;
  Co <- (expl.width/Nev)*c(-(1:Nev),(1:Nev));
  beta.id<-c(0,rep(idb,rep(2*Nev,length(idb))));

  if(cond.sse){
    icnt<-1;
    for(i in idb){
       freex<-free; freex[i]<-F;
       for(j in seq(Co)){
          bx<-fit$beta;
          bx[i]<-fit0$beta[i]+Co[j]*fit0$sd.beta[i];
          fit<-nlmap(Fo,prior,data,
                start=bx, free=freex,
                control=control);
          icnt<-icnt+1;
          SSE[icnt]<-fit$sse
          beta[,icnt]<-fit$beta
       }
    }
  } else {
    icnt<-1;
    for(i in idb){
       beta[,icnt+(1:(2*Nev))]<-fit0$beta;
       beta[i,icnt+(1:(2*Nev))]<-fit0$beta[i]+
                             Co*fit0$sd.beta[i];
       icnt<-icnt+2*Nev;
    }
    SSE<-calc.nlmap.sse(Fo,prior,beta,data)$sse;
  }

return(list(beta.id=beta.id,beta=beta,sse=SSE));
} # End of explore....


calc.nlmap.sse <- function(Fo,prior,beta,data){
```

```
#  Function to calculate SSEs ...
    if(!is.matrix(beta)) beta<-matrix(beta,length(beta),1);
    Y<-data$Y;
    sd.E<-data$sd.E;
    Np<-dim(beta)[1];
    Nobs<-length(Y);
    Npnt<-dim(beta)[2]

    Eb<-prior[,1];
    Wb<-1/prior[,2]^2;
    We<-1/sd.E^2;
    SSE<-numeric(Npnt);

# Note SSE is SSE-Nobs-Np
    for(i in 1:Npnt){
    SSE[i]<-sum(We*(Y-Fo(beta[,i],data=data)$Fo)^2-1) +
         sum(Wb*(beta[,i]-Eb)^2-1);
    }
    return(list(beta=beta,sse=SSE))
} # End clc.nlmap.sse
```

4 Monte Carlo Algorithms

4.1 ham.mc

This algorithm uses the "Hamiltonian" algorithm to produce Monte-Carlo simulations for constructing the "true" marginal posterior distribution of the regression model. This should solve exactly the same regression problem that *nlmpd* does. Here is its argument list;

```
ham.mc(Fo,Fdata,lg.prior,pdata,control=NULL,start=pdata$stats[,1])
```

4.1.1 Function Description

Inputs: The inputs *Fo*, *Fdata*, *lg.prior*, and *pdata* are exactly as described in Section 2.2. Here are descriptions of the remaining arguments;

control: This is a list that has the following:

burn.in: The number of burn-in iterations the algorithm performs. (default is 1000).

Niter: The number of iterations used for estimation. (Default is 1e4).

beta.id: The index numbers of the beta parameters that we want to put in an output table. (Default is all components in beta vector).

thinning: The proportion of simulations that will be saved to the output table. (Default is 10 for every tenth simulation.

se: Asymptotic standard errors for each of parameters.

start: This is the starting value for β. Default is the mean of the prior distribution.

Outputs: Output is a list that contains the following;

stats: This is a table of $length(\beta)$ by 5 that contains statistics from the Monte Carlo; It contains (1) the mean of beta, (2) the standard deviation, (3) the posterior mode, (4) the min, and (5) the max. In the future, this table might be expanded to include quantiles.

beta: This contains M.C. results for the parameters selected by *control*$*beta.id*.

lg.post: This contains the (scaled) log-posterior for each point recorded in *beta*.

diag: This would contain algorithm diagnostics, which would tell something about convergence status and the accuracy of the statistics presented in *stats*. Contains "bounces" and "metrop.rej"

Function Status: The algorithm has been tested on some standard problems. It is currently about 10 times slower than Metropolis Hastings and the resulting Markov Chain seems to have high correlations. The algorithm does work efficiently on distribution boundaries.

Also, the algorithm seems to get stuck at some values.

4.1.2 Algorithm Operation

During the Hamiltonian step, when a boundary is encountered, the appropriate momentums are reversed and the particle position is reflected.

4.1.3 S-PLUS Code

```
ham.mc <- function(Fo,Fdata,lg.prior,pdata,
              control=NULL,start=pdata$stats[,1]){
# NOTE: Argument list needs to be updated:
# Arguments:
#   Fo: Function describing regression model, Fo(beta,data)
#   Fdata: structure containing everything required by Fo
#   lg.prior: compute log-prior and its derivative
#   pdata: contains data required for lg.prior.  Must contain
#   component stats, length(beta) by 4 table describing prior
#          (mean, sd, min, max)
#   control: control parameters for algorithm.
#   start: start vector for beta.
#
# Output: a list containing...
#   stats: mean, sd, mode, min, max for parameters...
#   beta: table of simulated beta values
#   bout: indexes of parameters in beta table.
#   control: copy of control par. from input
#   diag: Algorithm diagnostics
#        bounces: # of reflections performed
#        metrop.rej: # of metropolis rejections performed


ham.step<-function(x,Fo,Fdata,lg.prior,pdata,control){
# NOTE need to fix up free parameters...
sd<-control$sd;
prior<-pdata$stats;
free<-sd>0;
nf<-sum(free);
x.prop<-x;
m<-1/sd[free]^2;
p <- rnorm(nf)/sd[free];
# Remainder is Metropolis step
# dt <- runif(1,0,control$dt.max)
dt <- control$dt.max;
dt.over.m <- dt/m
bounces<-rep(0,length(x));
```

```
p.prop <- p + dt/2 *
    log.post(x,Fo,Fdata,lg.prior,pdata,der=T)$der[free];
# NOTE: distributions do not have to have positive widths...
dpr<- (prior[free,4]-prior[free,3]);
for (it in seq(control$num.leaps)) {
  # x.last <- x.prop;
  x.prop[free] <- x.prop[free] + dt.over.m * p.prop

  # Make boundaries reflective
  z<- (x.prop[free]-prior[free,3])/dpr;
  bounces[free]<- bounces[free]+ abs(floor(z));
  z<- z %% 2;
  uz<-z>1;
  z[uz]<- 2-z[uz];
  p.prop[uz]<- -p.prop[uz];
  x.prop[free] <- prior[free,3]+ z*dpr;
  # Divide by 2 on last iteration
  # if(any(x.prop<prior[,3] | x.prop>prior[,4])) browser();
  p.prop <- p.prop +
    dt/((it==control$num.leaps)+1)*
       log.post(x.prop,Fo,Fdata,lg.prior,pdata,der=T)$der[free];
} # End for

# Accept/reject proposal
Ho <- log.post(x,Fo,Fdata,lg.prior,pdata)$val;
H <- sum(p^2/m)/2 - Ho
Hoo<-log.post(x.prop,Fo,Fdata,lg.prior,pdata)$val;
H.prop <- sum(p.prop^2/m)/2 - Hoo;

if (rexp(1) > H - H.prop) {
   x <- x.prop
   p <- p.prop
   metrop.rej<-0;
   Ho<-Hoo
} else {metrop.rej <- 1}

return(list(x.next=x,p.next=p,
            lpost=Ho, metrop.rej=metrop.rej,
    bounces=bounces));
} # End of Function.
```

```
log.post <- function(beta,Fo,Fdata,lg.prior,pdata,der=F){
   prior<-pdata$stats;
   free<-prior[,2]>0;
   # pdata<-list(stats=prior[free,]);
   xx<-lg.prior(beta,pdata,der=der);
   res<-Fo(beta,Fdata,der=der);
   lgp<-  -.5*sum(((Fdata$Y-res$Fo)/Fdata$sd.E)^2 -1) +
             sum(xx$lpr,na.rm=T);
   if(der){
     df<-rep(0,length(beta));
     df[free]<- apply(((Fdata$Y-res$Fo)/
                  Fdata$sd.E^2)*res$dF[,free],2,sum) +
                   xx$dlpr[free];
   } else{ df<-NULL }
   return(list(val=lgp,der=df))
} # End of log.post
```

```
old.seed<-.Random.seed;
prior<-pdata$stats;
if(is.null(control)){
  control<-list(
     burn.in=100, # burn-in iterations
     Niter=5000,  # # MC simulations
     beta.out=seq(dim(prior)[1]), # parameters to save
     thinning=1,
     sd=prior[,2], # Or set to asymptotic sd.
     num.leaps=10,
     dt.max=.025
  );
}

free<-prior[,2]>0;
control$sd[!free]<-0;
bout<-control$beta.out;
tser<-!is.null(bout);
b1<-start;
b1[!free]<-prior[!free,1];

# Do burnin:
for (i in 1:control$burn.in) {
```

```
res<-ham.step(b1,Fo,Fdata,lg.prior,pdata,control)
b1<-res$x.next;
}

Ns<-control$Niter;

beta.stats<-matrix(0,dim(prior)[1],5);
beta.stats[,3]<-b1;
beta.stats[,4]<-b1;
beta.stats[,5]<-b1;
bname<-dimnames(prior)[[1]]
dimnames(beta.stats)<-list(bname,c("mean","stdev","mode","min","max"));
if(tser){
    Nt<-control$thinning;
    bres <- matrix(0,length(bout),ceiling(Ns/ Nt));
    lpost<-numeric(ceiling(Ns/Nt));
    dimnames(bres)<-list(bname[bout],NULL);
}

bo<-b1;
mlp<- -Inf;
bounces<-rep(0,length(start));
metrop.rej<-0;

for (i in 1:Ns) {
    res<-ham.step(b1,Fo,Fdata,lg.prior,pdata,control);
    metrop.rej<-metrop.rej+res$metrop.rej;
    bounces<-bounces+res$bounces;
    b1<-res$x.next;
    beta.stats[,1]<-beta.stats[,1]+b1;
    beta.stats[,2]<- beta.stats[,2]+(b1-bo)^2;
    if(res$lpost>mlp){
      beta.stats[,3]<- b1;
      mlp<-res$lpost;
    }
    if(tser){
        ii<-(i+(Nt-1)) %/% Nt;
        bres[,ii]<-b1[bout];
lpost[ii]<-res$lpost;
    }

    beta.stats[,4]<-pmin(beta.stats[,4],b1);
    beta.stats[,5]<-pmax(beta.stats[,5],b1);
```

```
} # End for

beta.stats[,1]<-beta.stats[,1]/Ns;
beta.stats[,2]<- sqrt( pmax(beta.stats[,2]/Ns -
 (beta.stats[,1]-bo)^2,0));
diagn<-list(bounces=bounces,rej=metrop.rej);
if(tser) {
  return(list(stats=beta.stats,beta=bres,bout=bout,lpost=lpost,
         diag=diagn,control=control,seed=old.seed))
} else {
  return(list(stats=beta.stats,diag=diagn,control=control,seed=old.seed))
}
} # End of Function
```

4.2 metrop.mc

This algorithm uses the metropolis-hastings algorithm to produce Monte-Carlo simulations for constructing the "true" marginal posterior distribution of the regression model. This should solve exactly the same regression problem that *nlmpd* does. All details of program operation have not yet been worked out. Here is its argument list;

```
metrop.mc(Fo,Fdata,lg.prior,pdata,cov,start=pdata$stats[,1],control=NULL)
```

4.2.1 Function Description

Inputs: The inputs *Fo*, *Fdata*, *lg.prior*, and *pdata* are exactly as described in Section 2.2. Here are descriptions of the remaining arguments;

cov: This is the covariance matrix used for the algorithm's random walk.

start: This is the starting value for β. Default is the mean value of the prior distribution.

control: This is a list that has the following:

burn.in: The number of burn-in iterations the algorithm performs. (default is 1000).

Niter: The number of iterations used for estimation. (Default is 1e4).

beta.id: The index numbers of the beta parameters that we want to put in an output table. (Default is all components in beta vector).

thinning: The proportion of simulations that will be saved to the output table. (Default is 10 for every tenth simulation.

Outputs: Output is a list that contains the following;

stats: This is a table of $length(\beta)$ by 5 that contains statistics from the Monte Carlo; It contains (1) the mean of beta, (2) the standard deviation, (3) the posterior mode, (4) the min, and (5) the max. In the future, this table might be expanded to include quantiles.

beta: This contains M.C. results for the parameters selected by *control*$*beta.id*.

lg.post: This contains the (scaled) log-posterior for each point recorded in *beta*.

diag: This would contain algorithm diagnostics, which would tell something about convergence status and the accuracy of the statistics presented in *stats*. What would be in here has not been worked out exactly.

Function Status: Reflection at distribution boundaries was recently added to the MH algorithm. Reflection improves operation when parameters are on the prior distribution boundaries, but the rejection rate is still above 50%.

4.2.2 Algorithm Operation

Algorithm operation is fairly straight-forward; In iteration i, the algorithm produces β_{new} by sampling from a normal with mean β_i and covariance of cov. The ratio

$$\alpha = \frac{f_{post}(\beta_{new})}{f_{post}(\beta_i)} \tag{16}$$

is calculated and compared to a uniform $(0, 1)$ variate, U. If $U < \alpha$, then β_{new} is accepted for β_{i+1}. Otherwise, β_{i+1} is set to β_i.

When β_{i+1} is outside the hypercube, the point is "reflected" so that a point inside is obtained. The algorithm is run for a certain number of burn-in iterations, and then the statistics present in the *stats* table are produced.

4.2.3 S-PLUS Code

```
metrop.mc <- function(Fo,Fdata,lg.prior,pdata,cov,
             control=NULL, start=pdata$stats[,1]){

old.seed<-.Random.seed;
prior<-pdata$stats;
if(is.null(control))
control<-list(
          burn.in=1000, #  burn-in iterations
          Niter=5000,     #  number of MC iterations
          beta.out=seq(dim(prior)[1]),
  # ids of parameters to be tabulated
  thinning=1
         );


log.post <- function(beta,Fo,Fdata,lg.prior,pdata){
prior<-pdata$stats;
lpr<-lg.prior(beta,pdata)$lpr;
if(any(lpr== -Inf)){
    return(-Inf)
} else {
  return( -.5*sum( ((Fdata$Y-Fo(beta,Fdata)$Fo)/Fdata$sd.E)^2 -1) +
           sum( lpr )
        );
}
} # End of log.post....

free<-prior[,2]>0
bout<-control$beta.out;
```

```
tser<-!is.null(bout);
b1<-start;
lp1<-log.post(b1,Fo,Fdata,lg.prior,pdata);
b2<-b1

for (i in 1:control$burn.in) {
    b2[free] <- rmvnorm(1,b1[free],cov=cov);
    lp2<-log.post(b2,Fo,Fdata,lg.prior,pdata);
    if(runif(1) <= exp(lp2-lp1) ) {
              b1[] <- b2;
      lp1<-lp2;
    }
}

Ns<-control$Niter;

beta.stats<-matrix(0,dim(prior)[1],5);
beta.stats[,3]<-b1;
beta.stats[,4]<-b1;
beta.stats[,5]<-b1;
bname<-dimnames(prior)[[1]]
dimnames(beta.stats)<-list(bname,c("mean","stdev","mode","min","max"));
if(tser){
   Nt<-control$thinning;
    bres <- matrix(0,length(bout),ceiling(Ns/Nt));
    lpost<-numeric(ceiling(Ns/Nt));
    dimnames(bres)<-list(bname[bout],NULL);
}

bo<-b1;
mlp<-lp1;
mrej<-0;
dpr<-prior[free,4]-prior[free,3];
bounces<-rep(0,length(b1));
for (i in 1:Ns) {
    b2[free] <- rmvnorm(1,b1[free],cov=cov);

    # Do reflections....
    z<- (b2[free]-prior[free,3])/dpr;
    bounces[free]<-bounces[free]+abs(floor(z));
    z<- z %% 2;
    uz <- z>1;
    z[uz]<-2-z[uz];
```

```
    b2[free] <- (prior[free,3]+z*dpr);

    lp2<-log.post(b2,Fo,Fdata,lg.prior,pdata);
    if(rexp(1) >= lp1-lp2 ) {
              b1[] <- b2;
      lp1<-lp2;
    } else { mrej<-mrej+1 }
    if(tser){
        ii<-(i+(Nt-1)) %/% Nt;
        bres[,ii]<-b1[bout];
lpost[ii]<-lp1;
    }
    beta.stats[,1]<-beta.stats[,1]+b1;
    beta.stats[,2]<- beta.stats[,2]+(b1-bo)^2;
    if(lp1>mlp){
      beta.stats[,3]<- b1;
      mlp<-lp1
    }

    beta.stats[,4]<-pmin(beta.stats[,4],b1);
    beta.stats[,5]<-pmax(beta.stats[,5],b1);
}

beta.stats[,1]<-beta.stats[,1]/Ns;
beta.stats[,2]<- sqrt( beta.stats[,2]/Ns - (beta.stats[,1]-bo)^2);

diagn<-list(bounces=bounces,rej=mrej);
if(tser) {
  return(list(stats=beta.stats,beta=bres,
    bout=bout,lpost=lpost,diag=diagn,control=control,seed=old.seed))
} else {
  return(list(stats=beta.stats,diag=diagn,control=control,seed=old.seed))
}
} # End of Function
```

5 Monte Carlo Diagnostics

Diagnostics are useful for determining if the Markov chain simulations have converged to the stationary distribution and if the model being used is appropriate. The general approach to monitoring the convergence of the Markov chain simulations is based on multiple parallel sequences started at widely dispersed points of the target distribution.

One option for performing diagnostics is to use the CODA package of S-PLUS functions provided with the WINBUGS MCMC software. We plan to provide S-PLUS versions of various diagnostics that are immediately compatible with our input and output objects. The first such diagnostic is the potential scale reductions given below. A single Markov chain diagnostic due to Geweke (1992) is also given.

5.1 Potential Scale Reduction

This function calculates "estimated potential scale reductions" for every parameter in the Markov chain simulation. If the potential scale reduction is high, then the inference may be improved by running the simulation longer. As the simulation converges, the parallel chains will overlap and the potential scale reduction declines to 1. The potential scale reduction represents the factor by which potential Bayesian credible intervals will shrink once convergence is obtained (Gilks et al 1996). A potential scale reduction of 1.1 or less suggests sufficient convergence. Here is its argument list:

```
psr(stats,control)
```

5.1.1 Function Description

Inputs: The inputs *stats* and *control* are exactly as described in Section 2.2.

Outputs: The output is a vector of estimated potential scale reductions, a value for each parameter in the Markov chain simulation.

Function Status: This function checks that *stats* is a three-dimensional array of statistics from the multiple parallel Markov chain simulations. If only one simulation was performed, the function returns NAs.

5.1.2 S-PLUS Code

```
psr <- function(stats,control) {
   ds <- dim(stats)
   np <-ds[1]
   if(length(ds) == 2) return (rep(NA,np))
   if(ds[3] == 1) return (rep(NA,np))
   ni <- control$Niter
   B.vec <- colVars(t(stats[,1,]))*ni
   W.vec <- colMeans(t(stats[,2,])^2)
```

```
   vs.vec <- ((ni-1)/ni) * W.vec + (1/ni)*B.vec
   prs.vec <- sqrt(vs.vec/W.vec)
   return(prs.vec)
}
```

5.2 Spectral Density Diagnostic

This function calculates Geweke's (1992) spectral density diagnostic Z_n. This diagnostic takes two sections of a single Markov chain simulation and tests for equal location. Hence, this diagnostic can only indicate that convergence has not been achieved. If $\mid Z_n \mid > 2.0$, then the simulation should be run longer. Here is its argument list:

```
geweke(beta,frac1=0.1,frac2=0.5)
```

5.2.1 Function Description

Inputs: The input *beta* is an output from metrop.mc.

Outputs: The output is a vector of Geweke convergence diagnostics, a value for each parameter in the Markov chain simulation.

Function Status: This function has default partitions of one-tenth and one-half as suggested by Geweke (1992). *geweke* calls *geweke.power*, which calculates the spectral sensitivity estimates. Control of these could be given to the user.

5.2.2 S-PLUS Code

```
geweke <- function(beta,frac1=0.1,frac2=0.5) {
        ni <- dim(beta)[2]
        np <- dim(beta)[1]
        gcd <- rep(NA,np)
        for(i in 1:np) {
        x <- beta[i,]
        power1 <- geweke.power(x[1:(ni * frac1)])
        power2 <- geweke.power(x[(ni - (ni * frac2) + 1):ni])
        bbara <- mean(x[1:(ni * frac1)])
        bbarb <- mean(x[(ni - (ni * frac2) + 1):ni])
        gcd[i] <- (bbara - bbarb)/sqrt((power1/(ni * frac1)) +
                                    (power2/(ni * frac2)))
        }
        gcd
}

geweke.power <- function(x) {
        ni <- length(x)
```

```
        nspans <- sqrt(ni)/0.3 + 1
        if(nspans >= ni) nspans <- ni - 1
        pgram1 <- spec.pgram(x,spans=nspans,demean=T,detrend=F,plot=F)
        power <- (10^(pgram1$spec[1]/10))
        power
}
```

6 IR Modeling Functions

This section contains the basic functions used to simulate data from a 3-layer model for LWIR measurements from a passive instrument; see Beer (1991) or Milman (1999). The parameters for this 3-layer model are

$$\beta = (\mathcal{C}_1, \mathcal{C}_2, ..., T_g, T_p, T_a, \alpha_1, \alpha_2, ...), \tag{17}$$

where $\mathcal{C}_i$ are the gas concentrations; T_g, T_p, and T_a are the temperatures of the ground, plume, and atmosphere, respectively; and α_i are the emissivity loadings. The ground spectrum is;

$$R_g = e_g \mathcal{B}(T_g), \tag{18}$$

where $\mathcal{B}(\cdot)$ is the Planck function and the ground emissivity is defined by;

$$e_g = AvE + dE\alpha. \tag{19}$$

The plume spectrum is;

$$R_p = \tau_p R_g + (1 - \tau_p)\mathcal{B}(T_p). \tag{20}$$

The atmospheric spectrum is;

$$R_a = \tau_a R_p + (1 - \tau_a)\mathcal{B}(T_a). \tag{21}$$

The plume transmissivity is defined by;

$$\tau_p = \exp(-\sum_j A_j \mathcal{C}_j). \tag{22}$$

The atmospheric transmissivity τ_a is an input to the modeling.

6.1 Plancka

This function calculates a Planck function and if desired, the first and second derivative with respect to temperature.

```
Plancka(nu,Tk,der=0,units="watts")
```

nu: a vector of wave numbers (1/cm).

Tk: a vector of temperatures in Kelvin. Note: either nu or Tk can be a single value and in this case, it is expanded to match the length of the other vector.

der: Specifies what's to be calculated; der=0 produces the Planck function, der=1, the first derivative with respect to Tk, and der=2 the second derivative.

units: *watts* produces results in watts per (steridian cm^2 wavenumber). *photons* produces results in photons per (steridian cm^2 wavenumber). *wavenumber* is in cm^{-1}.

Output: Is a vector of same length as max(nu,Tk). The vector contains either Planck function, or its derivative.

6.1.1 S-PLUS Code

```
Plancka <- function(nu, Tk, der=0,units = "watts") {
# DESCRIPTION: Calculates Plancks Function for a set of
#   wavenumbers and temperatures. Can also calculate the
#   first and second derivatives with respect to temperature
# ARGUMENTS:
#   nu: a vector of wave numbers (cycles/cm)
#   Tk: a vector of temperatures in Kelvin
#       (note nu and Tk may be of different lengths,
#        and in this case, the shorter cycles)
#   der: specifies whats to be calculated; der=0 produces the
#        Planck function, der=1, the first derivative, and
#        der=2, the second derivative.
#   units: Specifies what units for output; "watts" produces
#       values in watts per (steridian cm^2 wavenumber),
#       "photons" produces photons per (steridian cm^2 waveno).
# VALUE: is a vector of max(length(nu),length(Tk)) containing
#        the Planck values.

  if(!any(units == c("photons", "watts")))
 stop("units must be either watts or photons");
  if(is.na(match(der[1],c(0,1,2))))
         stop("der must be either 0,1, or 2");

  # Planck's constant
  h <- 6.62606876e-34
  # Boltzman's constant
  K <- 1.3806503e-23
  # Speed of light
  co <- 29979245800.
  if(units == "photons") Uo <- 1.  else Uo<-co*h*nu;
  C1 <- 2. * co * nu^2.
  C2 <- (h * co * nu)/K
  U <- exp(C2/Tk)
  if(der==0)
     B<-C1/(U-1.);
  if(der==1)
     B<-(C1*C2*U)/(Tk*(U - 1.))^2.;
  if(der==2)
     B<-(C1*C2*U*(C2*(1.+U)-2.*Tk*(U-1.)))/(Tk^4.*(U-1.)^3.);
  return(Uo*B)
} # End of Function
```

6.2 simulator

This function simulates IR spectra obtained from a passive instrument looking at a gas plume through the atmosphere.

```
simulator(scen,Nsim,resol=NA,debug=F)
```

scen: A list containing information that defines the simulation scenario. Here are its components;

nu: Wave-number vector.

abs.spectra: A table containing gas absorbance spectra in the plume.

tau.atmos: Atmospheric transmissivity.

emiss.ground: A table containing the ground emissivities.

gas.distr: A table containing the gas concentration distributions. Each row represents a gas and the row contains the distribution min and max. The gase concentrations are sampled from a uniform distribution.

temp.distr: A table containing the ground, plume and atmospheric temperatures, which are assumed to be normally distributed. The table contains the distribution means and stdev's.

ground.distr: A data frame describing the ground materials. It must contain the component "dist" which contains the multinomial probs used in the simulation.

stdev.instr: A vector containing the stdev's associated with each wavenumber.

Nsim: The number of simulations to produce.

resol: If resol=NA, the scenario resolution is not changed. Otherwise, the results are averaged over a window of length *resol* and the vector step size is changed to *resol*. All associated vectors and matricies in *scen* are altered to reflect the new resolution.

debug: If true, spectra for ground, plume, and atmosphere are produced.

Output: A list that contains;

scenario: The input *scen*.

misc: All of the intermediate simulated values, such as e_g, T_g, T_p, T_a, and the gas-concentrations.

sim.spectra: A table of the simulated spectra.

6.2.1 S-PLUS Code

```
simulator<-function(scen,Nsim,resol=NA,debug=F){
# DESCRIPTION: This function simulates an IR radiance spectra from a
# single pixel.  The model used is a simple 3-layer model (ground,
# plume, and atmosphere). Random variability in the model is
#  1. temperatures of the three layers (normal)
#  2. ground emissivity (a multinomial sample from emissivity library)
#
#  This has been altered to change the resolution...
#
# ARGUMENTS:
#   scen: A list containing all input parameters that define the
#     scenario. Here are its components:
#     nu: Vector of wavenumbers used
#     abs.spectra: A length(nu) by N table containing gas abs spectra.
#     tau.atmos: a vector of length(nu) containing the atmospheric
#                transmissivity
#     emiss.ground: A table containing the ground emissivities
#     gas.distr: A table, with each row describing a gas. Must have
#                the components "dist.low", and "dist.hi", which
#         define the boundaries of the gas.distr priors.
#     temp.distr: A table containing "mean" and "stdev" as columns
#                 and rows "Tg","Tp","Ta".
#     ground.distr: A table that contains a description of the ground
#                   materials. Must have the component "dist" which
#    contains the multinomial probs used in simulation.
#     stdev.instr: A vector of length(nu) describing the stdev at
#                  each spectral bin.
#   Nsim: The number of simulations to run.
#
#   resol: If NA, the input resolution is not changed. Otherwise, final
#   results are convolved with a window of width "resol". All associated
#   spectra and matricies in scenario are changed. The instrument error
#   is also convolved.
#
# VALUE: A list that contains the following:
#    scenario: the input argument "scen"
#    misc: Some of the intermediate simulated values. These
#          include Kg,Tg,Tp,Ta, and gas-concentrations.
#    sim.spectra: A table of dim length(nu) by Nsim containing
#          the simulated spectra.

Planck2<-function(nu,Tk){
```

```
#  outer product Planck function:
    res<-array(0,dim=c(length(nu),length(Tk)));
    for(i in 1:length(Tk)){
     res[,i]<-Plancka(nu,Tk[i])
    }
    return(res)
} # End of Planck2

nu<-scen$nu;
Tdist<-scen$temp.dist;
Edist<-scen$ground.distr$dist;
tau.a<-as.vector(scen$tau.atmos);

# simulate random parameters....
# generate ground emissivities..
Kg<-sample(1:dim(scen$emiss.ground)[2],
           size=Nsim,replace=T,
           prob=Edist);

# generate temperatures....
Tg<-Tdist[1,1]+Tdist[1,2]*rnorm(Nsim)
Tp<-Tdist[2,1]+Tdist[2,2]*rnorm(Nsim);
Ta<-Tdist[3,1]+Tdist[3,2]*rnorm(Nsim);

# generate plume transmissivities...
beta<-array(0,dim=c(dim(scen$gas.distr)[1],Nsim));
lo<-scen$gas.distr$dist.lo;
hi<-scen$gas.distr$dist.hi;
for(i in 1:Nsim) beta[,i]<-lo+(hi-lo)*runif(dim(beta)[1]);
dimnames(beta)<-list(paste("gas",1:dim(beta)[1],sep="."),NULL);
tau.p<-exp(- scen$abs.spectra %*% beta);

# assemble generated random variables for output...
misc<-cbind(Kg=Kg,Tg=Tg,Tp=Tp,Ta=Ta,t(beta))

# Now calculate spectra...
if(debug){
   # simulate ground radiance....
   Rg<-scen$emiss.ground[,Kg]*Planck2(nu,Tg);

   # simulate plume layer...
   Rp <- tau.p*Rg + (1-tau.p)*Planck2(nu,Tp);
```

```
   # simulate atmosphere...
   Ra <- tau.a*Rp + (1-tau.a)*Planck2(nu,Ta);

   # simulate instrument noise...
   Rm <-  Ra + outer(scen$stdev.instr,rep(1,Nsim))*
           array(rnorm(Nsim*length(nu)),dim=dim(Rg));

   # here is code to alter resolution...
   if(!is.na(resol)){
      nu.out<-seq(from=min(nu),to=max(nu),by=resol);
      scen$nu<-nu.out;
      scen$tau.atmos<-Average2(nu,tau.a,nu.out,resol);
      scen$emiss.ground<-Average2(nu,scen$emiss.ground,nu.out,resol);
      scen$abs.spectra<-Average2(nu,scen$abs.spectra,nu.out,resol);
      scen$stdev.instr<-sqrt(Average2(nu,scen$stdev.instr^2,nu.out,resol));
      Rm<-Average2(nu,Rm,nu.out,resol);
   }
   return(list(scenario=scen,misc=misc,
          Rg=Rg,Rp=Rp,Ra=Ra,sim.spectra=Rm))
}else{
   # simulate ground radiance....
   Rm<-scen$emiss.ground[,Kg]*Planck2(nu,Tg);

   # simulate plume layer...
   Rm[]<- tau.p*Rm + (1-tau.p)*Planck2(nu,Tp);

   # simulate atmosphere...
   Rm[] <- tau.a*Rm + (1-tau.a)*Planck2(nu,Ta);

   # simulate instrument noise...
   Rm[] <-  Rm + outer(scen$stdev.instr,rep(1,Nsim))*
           array(rnorm(Nsim*length(nu)),dim=dim(Rm));

   # here is code to alter resolution...
   if(!is.na(resol)){
      nu.out<-seq(from=min(nu),to=max(nu),by=resol);
      scen$nu<-nu.out;
      scen$tau.atmos<-Average2(nu,tau.a,nu.out,resol);
      scen$emiss.ground<-Average2(nu,scen$emiss.ground,nu.out,resol);
      scen$abs.spectra<-Average2(nu,scen$abs.spectra,nu.out,resol);
      scen$stdev.instr<-sqrt(Average2(nu,scen$stdev.instr^2,nu.out,resol));
      Rm<-Average2(nu,Rm,nu.out,resol);
   }
```

```
   return(list(scenario=scen,misc=misc,sim.spectra=Rm))
}
} # End of Function...
```

6.3 F.3layer

This is one version of our 3-layer regression model, the other is $F.3layer.mix$. The 3 layers are the ground, the plume, and the atmosphere.

`F.3layer(beta,data)`

beta: Vector of unknown parameters. Beta consists of (1) plume gas concentrations, (2) layer temperatures, and (3) emissivity loadings.

data: This is a list that contains all other information required in the calculation. Here are the components;

nu: Vector of wavenumbers.

Abs: Table of gas absorbance spectra.

tau.a: Vector of atmospheric transmissivity.

AvE: Average ground emissivity.

dE: Eigen-vectors of ground emissivity.

Output: Contains Fo, the calculated spectra, and dF, the derivative with respect to beta.

6.3.1 S-PLUS Code

```
F.3layer <- function(beta,data,der=T){
# DESCRIPTION: This function evaluates the 3-layer
#   regression model and calculates the derivatives
#   with respect to the parameter vector, beta.
# ARGUMENTS:
#   beta: vector of unknown parameters. beta consists
#         of plume gas concentrations, layer temperatures,
#  and emissivity eigenvector loadings.
#   data: This is a list that contains all other known
#     information appearing in the 3-layer model. Here
#     are the components of the list:
#       nu: vector of wavenumbers.
#       Abs: Matrix length(n) by ??? of absorbance spectra.
#       tau.a: Atmospheric transmissivity.
#       AvE: Average ground emissivity.
#       dE: Eigen-vectors of ground emissivity.
# VALUE: is a list containing the evaluated result and the
#       derivative.
#       Fo: evaluated result, vector of length(nu)
#       dF: derivatives, matrix of length(nu) by length(beta)
# DETAILS:
```

```
#  The three layers are the ground (g), the plume (p), and the
#  atmosphere (a). The radiance spectrum at the top of each of
#  these layers is Rg, Rp, Ra, respectively. Let:
#            Eg: Emissivity of ground
#     Bg,Bp,Ba: Blackbody radiation of each layer
#     tau.a, tau.p: transmissivity of atmosphere and
#           plume
#  Then the 3-layer model is described by;
#        Rg = Eg*Bg
#        Rp = tau.p*Rg + (1-tau.p)*Bp
#        Ra = tau.a*Rp +(1-tau.a)*Ba
  nu<-data$nu;
  tau.a<-as.vector(data$tau.a);
  Nnu<-length(nu);
  Ng<-dim(data$Abs)[2];
  Nt<-3;
  Nalph<-dim(data$dE)[2];
  gid<-1:Ng;
  tid<-Ng+(1:Nt);
  eid<-Ng+Nt+(1:Nalph);
  Tkel<-beta[tid];
  BB<-cbind(
        Plancka(nu,Tkel[1]),
        Plancka(nu,Tkel[2]),
        Plancka(nu,Tkel[3]));

  # calculate value
  Eg <- as.vector(
         data$AvE + data$dE %*% beta[eid]);
  Rg <-  Eg*BB[,1];
  tau.p<-as.vector(
         exp(- data$Abs %*% beta[gid]));
  Rp <-  tau.p*Rg + (1-tau.p)*BB[,2];
  Rm <-  tau.a*Rp + (1-tau.a)*BB[,3];
  if(der){
    dBB<-cbind(
        Plancka(nu,Tkel[1],der=1),
        Plancka(nu,Tkel[2],der=1),
        Plancka(nu,Tkel[3],der=1));
    dFgas <- (tau.a*(BB[,2]-Rg)*tau.p)*data$Abs;
    dFT <- cbind(
           tau.a*tau.p*Eg*dBB[,1],
           tau.a*(1-tau.p)*dBB[,2],
```

```
        (1-tau.a)*dBB[,3]);
  dFalph <- tau.a*tau.p*BB[,1]*data$dE;
  return(list(Fo=as.vector(Rm),dF=cbind(dFgas,dFT,dFalph)))
} else{
  return(list(Fo=as.vector(Rm),dF=NULL))
}
} # End of Function
```

6.4 F.3layer.mix

`F.3layer.mix(beta,data,der=T)`

This does the same calculation as $F.3layer$, except that the ground emissivity is a mixture of the emissivities stored in dE. The difference is;

$$e_g = dE\alpha' \tag{23}$$

where

$$\alpha'_i = \frac{\alpha_i}{\alpha_+} \tag{24}$$

6.4.1 S-PLUS Code

```
F.3layer.mix <- function(beta,data,der=T){
# DESCRIPTION: This function evaluates the 3-layer
#   regression model and calculates the derivatives
#   with respect to the parameter vector, beta.
# ARGUMENTS:
#   beta: vector of unknown parameters. beta consists
#         of plume gas concentrations, layer temperatures,
#  and emissivity eigenvector loadings.
#   data: This is a list that contains all other known
#     information appearing in the 3-layer model. Here
#     are the components of the list:
#       nu: vector of wavenumbers.
#       Abs: Matrix length(n) by ??? of absorbance spectra.
#       tau.a: Atmospheric transmissivity.
#       AvE: Average ground emissivity.
#       dE: Eigen-vectors of ground emissivity.
# VALUE: is a list containing the evaluated result and the
#       derivative.
#       Fo: evaluated result, vector of length(nu)
#       dF: derivatives, matrix of length(nu) by length(beta)
# DETAILS:
#  The three layers are the ground (g), the plume (p), and the
#  atmosphere (a). The radiance spectrum at the top of each of
#  these layers is Rg, Rp, Ra, respectively. Let:
#            Eg: Emissivity of ground
#     Bg,Bp,Ba: Blackbody radiation of each layer
#     tau.a, tau.p: transmissivity of atmosphere and
#           plume
#  Then the 3-layer model is described by;
#        Rg = Eg*Bg
#        Rp = tau.p*Rg + (1-tau.p)*Bp
```

```
#        Ra = tau.a*Rp +(1-tau.a)*Ba
  nu<-data$nu;
  tau.a<-as.vector(data$tau.a);
  Nnu<-length(nu);
  Ng<-dim(data$Abs)[2];
  Nt<-3;
  Nalph<-dim(data$dE)[2];
  gid<-1:Ng;
  tid<-Ng+(1:Nt);
  eid<-Ng+Nt+(1:Nalph);
  Tkel<-beta[tid];
  BB<-cbind(
        Plancka(nu,Tkel[1]),
        Plancka(nu,Tkel[2]),
        Plancka(nu,Tkel[3]));

  # calculate value
  apl<-sum(beta[eid]);
  Eg <- as.vector(
         data$AvE + data$dE %*% beta[eid]/apl);
  Rg <-  Eg*BB[,1];
  tau.p<-as.vector(
         exp(- data$Abs %*% beta[gid]));
  Rp <-  tau.p*Rg + (1-tau.p)*BB[,2];
  Rm <-  tau.a*Rp + (1-tau.a)*BB[,3];
  if(der){
    dBB<-cbind(
        Plancka(nu,Tkel[1],der=1),
        Plancka(nu,Tkel[2],der=1),
        Plancka(nu,Tkel[3],der=1));
    dFgas <- (tau.a*(BB[,2]-Rg)*tau.p)*data$Abs;
    dFT <- cbind(
           tau.a*tau.p*Eg*dBB[,1],
           tau.a*(1-tau.p)*dBB[,2],
           (1-tau.a)*dBB[,3]);
    dFalph <- tau.a*tau.p*BB[,1]*data$dE *
            outer(rep(1,Nnu),(apl-beta[eid])/apl^2);
    return(list(Fo=as.vector(Rm),dF=cbind(dFgas,dFT,dFalph)))
  } else{
    return(list(Fo=as.vector(Rm),dF=NULL))
  }
  } # End of Function
```

6.5 Average2

`Average2(nu,U,nu.out,dnu.out)`

The function changes resolution on an arbitrary spectrum. If *dnu.out* is smaller than the minimum difference in nu, this does interpolation. Otherwise it averages over a window of width dnu.out around every output point.

nu: vector containing wavenumbers for the input spectrum.

U: Vector or matrix containing the spectra. If matrix, every column is a spectra.

nu.out: Vector of wavenumbers for which we want spectral values.

dnu.out: The output resolution. To calculate the value associated with nu.out[i], average over a window of width dnu.out.

Output: Is a matrix of dim $length(nu.out)$ by $dim(U)[2]$.

6.5.1 S-PLUS Code

```
Average2 <- function(nu, U, nu.out, dnu.out) {
# DESCRIPTION: Function to change resolution on an arbitrary
#  spectrum. If dnu is smaller than input spectra, this does
#  interpolation. If dnu is larger, it averages.
# ARGUMENTS:
#  nu: vector containing wavenumbers for input spect.
#  U: vector (or matrix) containing the spectra. If matrix,
#     every column is a spectra.
#  nu.out: vector of wavenumbers for which we want spectral
#     values.
#  dnu.out: The output resolution: to calculate the value
#     associated with nu.out[i], average all points within
#     the interval nu.out[i]+-dnu.out/2.
# VALUE: Is a matrix of dimension length(nu.out) by dim(U)[2].
#
# If values in nu.out are outside range(nu), then U[1] or
# U[last] are used.

  if(!is.matrix(U)) U <- array(U, dim = c(length(U), 1.))
  res <- array(0., dim = c(length(nu.out), dim(U)[2.]))
  dnu<-min(diff(nu));
  n <- (dnu.out/dnu) %/% 2.
  for(i in 1.:dim(U)[2.]) {
    for(j in ( - n):n)
      res[, i] <- res[, i] +
```

```
      approx(nu,U[,i],xout = nu.out+j*dnu, rule = 2.)$y
  }
  res <- res/(2. * n + 1.)
  if(dim(res)[2.] == 1.)
res <- as.vector(res)
  return(res)
} # End of Function
```

6.6 Extended Beta Distributions

Here are the extended beta distributions:

```
debeta(Y,pram=c(.5,1/sqrt(12),0,1))
```

This calculates the density for various points in the vector Y. *pram* contains four parameters, which specify the distribution, which are its mean, standard deviation, minimum and maximum (μ, σ, M_1, M_2). Here is the density formula for a point $M_1 < Y < M_2$:

$$f(Y) = \frac{1}{M_2 - M_1} \frac{\Gamma(a_1 + a_2)}{\Gamma(a_1)\Gamma(a_2)} \left(\frac{Y - M_1}{M_2 - M_1}\right)^{a_1 - 1} \left(\frac{M_2 - Y}{M_2 - M_1}\right)^{a_2 - 1} \tag{25}$$

with $(\mu - M_1)/(M_2 - M_1) = a_1/a_+$ and $a_+ = (\mu - M_1)(M_2 - \mu)/\sigma^2$.

Note: *pram* can be a matrix of $length(Y)$ by 4 and in this case different parameters are applied to each component in Y. Also, an error will result if impossible values are supplied for *pram*.

```
rebeta(N,pram=c(.5,1/sqrt(12),0,1))
```

This generates N random variates from the extended beta distribution.

```
lg.ebeta(beta,pdata,der=F)
```

This is a trivial variant of *debeta*, which produces the log-density required by the estimation functions.

6.6.1 S-PLUS Code

```
rebeta<-function(N,pram=c(.5,1/sqrt(12),0,1)){
# Function calculates N "extended beta" random
# variates from a distribution with parameters "pram".
# "pram" contains the mean, the standard deviation,
# the min, and the max. A standard deviation of 0
# is allowed.

   dC<-pram[4]-pram[3];
   mu<-(pram[1]-pram[3])/dC;
   sd<-pram[2]/dC;
   if(pram[2]==0){
      return(rep(pram[1],N))
   } else {
      Ap <- mu*(1-mu)/sd^2 -1;
      if(Ap<=0){
            print(pram)
            stop("rebeta parameters impossible \n");
```

```
      }
      A1 <- mu*Ap;
      A2 <- Ap-A1;
      return(pram[3]+dC*rbeta(N,A1,A2))
   }
} # End of rebeta



debeta<-function(Y, pram = c(0.5, 1/sqrt(12), 0, 1)) {
# Function calculates "extended beta" densities.
# at the values in the vector Y. "pram" contains
# the dist. parameters (mean,sd,min,max).  pram can
# either be a vector or matrix.  A standard deviation
# of 0, or impossible combination of parameters results
# in a returned value of "NA".
if(!is.matrix(pram)) pram <- t(matrix(pram, 4, length(Y)))
if(dim(pram)[1]!=length(Y)) stop("pram has incorrect row dimension\n")
dC <- pram[, 4] - pram[, 3];
mu <- (pram[, 1] - pram[, 3])/dC;
sd <- pram[, 2]/dC;
Ap <- (mu * (1 - mu))/sd^2 - 1;
u <- sd > 0 & Ap > 0
if(any(Ap <= 0)) warning("debeta parameters are impossible\n")
fo <- rep(NA, length(Y))
A1 <- mu[u] * Ap[u];
A2 <- Ap[u] - A1;
Yp <- (Y[u] - pram[u,3])/dC[u];
fo[u] <- dbeta(Yp, A1, A2)/dC[u];
return(fo)
} # End of debeta


lg.ebeta<-function(Y,pdata,der=T){
# Function calculates the log of the extended beta and its derivative
# with respect to the argument.

pram<-pdata$stats;
dC<-pram[,4]-pram[,3];
mu<-(pram[,1]-pram[,3])/dC;
sd<-pram[,2]/dC;
Ap <- mu*(1-mu)/sd^2 -1;
u<-sd>0 & Ap>0;
```

```
if(any(Ap<=0)) warning("debeta parameters are impossible\n");
lpr<-rep(NA,length(Y));
A1 <- mu[u]*Ap[u];
A2 <- Ap[u]-A1;
Yp<- (Y[u]-pram[u,3])/dC[u];
lpr[u] <- log(dbeta(Yp,A1,A2)/dC[u]);
if(der){
  dlpr<-lpr;
  dlpr[u]<- (A1-1)/(Y[u]-pram[u,3]) + (A2-1)/(Y[u]-pram[u,4]);
  dlpr[Y<=pram[,3] | Y>=pram[,4]]<-0;
  dlpr[!u]<-NA;
} else {
  dlpr<-NULL
}
return(list(lpr=lpr,dlpr=dlpr))
} # End of function
```

6.7 Matched Filter Estimators

The functions in this section perform "standard" matched filter estimation. One function, *bkgrnd.anal*, will do the principal components decomposition on the background to account for clutter. *single.filter* produces individual filters for a set of gases, while *mult.filter* produces a combined filter. Below is a brief description of each;

`bkgrnd.anal(bground)`

bground: is a matrix containing background spectra. The mean and covariance of these spectra are computed.

Output: is a list containing the *mean*, U, and D. U and D are the decomposed version of the covariance; $Cov(R) = Udiag(D^2)U^t$.

`single.filter(bg.stats,plume,scen)`

and

`mult.filter(bg.stats,plume,scen)`

bg.stats: Output from *bkgrnd.anal*.

plume: Matrix containing spectra.

scen: List containing parameters required by estimator. Matched filter uses *nu*, *abs.spectra*, *emiss.ground*, *temp.distr*, and *tau.atmos*.

6.7.1 S-PLUS Code

```
bkgrnd.anal<-function(bground){
  R.bgr<-apply(bground,1,mean);
  # decomposition is  X = U * diag(d) * t(V);
  res<-svd(bground-R.bgr, nv=0)
return(list(mean=R.bgr,U=res$u,D=res$d/sqrt(dim(bground)[2]-1)))
}



single.filter<-function(bg.stats,plume,scen){
# DESCRIPTION: This fits a matched filter regression, with model of
#
#      R.plume - R.bgr = tau.atmos*(Bp-e*Bg)*Ak*Betak) + E
#
#      where k indexes gases, R.bgr = mean(backgr) and
#      cov(E)= cov(backgr).  e stands for mean emissivity.
```

```
#
# ARGUMENTS:
#    bg.stats: is list of background mean and cov matrix;
#              (no plume).  Used to calculate average background and
#      error covariance. Cov matrix is stored as U*D*t(U).
#    plume:  is a matrix of length(nu) by ?? containing plume spectra
#            to be processed.
#    scen: This is list containing all parameters used to generate the
#          scene. matched filter uses; nu, abs.spectra, emiss.ground,
#   temp.distr, and tau.atmos.
#
# VALUE: contains est. stderr

nu<-scen$nu;
R.bgr<-bg.stats$mean
Cov<- (bg.stats$U %*% (bg.stats$D^2*t(bg.stats$U))) +
           diag(scen$stdev.instr^2);

e.av<-apply(scen$emiss.ground,1,mean);
tau.a<-scen$tau.atmos;
Bp<-Plancka(nu,scen$temp.distr[2,1]);
Bg<-Plancka(nu,scen$temp.distr[1,1]);
W<-as.vector(tau.a*(Bp-e.av*Bg));
X<-W*scen$abs.spectra;
Ngas<-dim(X)[2];
Nplume<-dim(plume)[2];

Z<-chol.solve(Cov,X);
stderr<-numeric(Ngas);
for(i in 1:Ngas) {
   stderr[i]<-1/sqrt(sum(Z[,i]*X[,i]));
   Z[,i]<-Z[,i]*stderr[i]^2;
}
Z<-t(Z);

# here is estimation....
est<- Z %*% (plume-R.bgr);
dimnames(est)<-list(dimnames(scen$abs.spect)[[2]],NULL);
names(stderr)<-dimnames(est)[[1]];
return(list(beta=est,sd.beta=stderr))
} # End of function
```

```
mult.filter<-function(bg.stats,plume,scen){
# DESCRIPTION: This fits a matched filter regression, with model of
#
#      R.plume - R.bgr = tau.atmos*(Bp-e*Bg)*(sum Ak*Betak) + E
#
#      where k indexes gases, R.bgr = mean(backgr) and
#      cov(E)= cov(backgr).  e stands for mean emissivity.
#
# ARGUMENTS:
#    bground: is matrix of length(nu) by M of background spectra.
#             (no plume).  Used to calculate average background and
#      error covariance.
#    plume:  is a matrix of length(nu) by ?? containing plume spectra
#            to be processed.
#    scen: This is list containing all parameters used to generate the
#          scene. matched filter uses; nu, abs.spectra, emiss.ground,
#   temp.distr, and tau.atmos.
#
# VALUE: contains est. stderr

nu<-scen$nu;
R.bgr<-bg.stats$mean;
Cov<- (bg.stats$U %*% (bg.stats$D^2*t(bg.stats$U))) +
           diag(scen$stdev.instr^2);

e.av<-apply(scen$emiss.ground,1,mean);
tau.a<-scen$tau.atmos;
Bp<-Plancka(nu,scen$temp.distr[2,1]);
Bg<-Plancka(nu,scen$temp.distr[1,1]);
W<-as.vector(tau.a*(Bp-e.av*Bg));
X<-W*scen$abs.spect;
Ngas<-dim(X)[2];
Nplume<-dim(plume)[2];

Z<-chol.solve(Cov,X);
Z<-t(Z);
Precis<- Z %*% X;
Precis<-(Precis +t(Precis))/2;

Cov.est<-chol.solve(Precis,diag(rep(1,Ngas)));
stderr<-sqrt(diag(Cov.est));
Z <- Cov.est %*% Z;
```

```
# estimate gases
est<- Z %*% (plume-R.bgr);
dimnames(est)<-list(dimnames(scen$abs.spect)[[2]],NULL);
names(stderr)<-dimnames(est)[[1]];
dimnames(Cov.est)<-list(names(stderr),names(stderr));

return(list(beta=est,sd.beta=stderr,Cov.est=Cov.est))
} # End of function
```

7 Bibliography

[1] "Acorn User's Guide," Version 3.11, Analytical Imaging and Geophysics LLC, 4450 Arapahoe Av. suite 100, Boulder, CO 80303, 2001.

[2] Beer R, **Remote Sensing by Fourier Transform Spectrometry,** Wiley Interscience, New York, 1991.

[3] Berger JO, **Statistical Decision Theory and Bayesian Analysis,** 2nd Ed, Springer Verlag, New York, 1980.

[4] Borel CC, "Surface Emissivity and Temperature Retrieval for a Hyperspectral Sensor," LANL, cborellanl.gov.

[5] Gelman A, Carlin JB, Stern HS, and Rubin DB, **Bayesian Data Analysis**, Chapman and Hall/CRC, New York, 2000.

[6] Geweke J, "Evaluating the Accuracy of Sampling-based Approaches to the Calculation of Posterior Moments," in JM Bernardo, AFM Smith, AP Dawid, and JO Berger (eds), **Bayesian Statistics 4**, Oxford University Press, New York, 1992.

[7] Gilks WR, Richardson S, and Spiegelhalter DJ, **Markov Chain Monte Carlo in Practice**, Chapman and Hall/CRC, New York, 1998.

[8] Harsanyi JC, "Hyperspectral Image Classification Dimensionality Reduction: An Orthogonal Subspace Projection Approach," IEEE Transactions on Geoscience and Remote Sensing, Vol 32, No 4, July 1994.

[9] "HIRIS Algorithm Physics Model Description Version 2.3," Lawrence Livermore National Laboratory, Livermore, CA, June 2, 2000.

[10] Hanson K, "Markov Chain Monte Carlo Posterior Sampling with the Hamiltonian Method," Proc. 3rd Int. Symposium on Sensitivity Analysis of Model Output, 2001.

[11] Milman AS, **Mathematical Principles of Remote Sensing**, Sleeping Bear Press, Chelsea MI, 1999.

[12] Spiegelhalter D, Thomas A, and Best N, "WinBUGS Version 1.3 User Manual," http://www.mrc-bsu.cam.ac.uk/bugs, April 2000.

[13] Westervelt R., "Imaging Infrared Detectors II," JSR-97-600, Jason Com., MITRE Corp, Oct. 2000.