Hands-On 9

Inverse Uncertainty Quantification: Parameter Estimation

Exercise 1

Let us consider a simple nonlinear model with two parameters,

$$y = \theta_1(1 - \exp(-\theta_2 x)).$$

The model is used to describe the biological oxygen demand (BOD). The goal is to estimate the parameters and their uncertainty using the data $\mathbf{x} = (1, 3, 5, 7, 9)$ and $\mathbf{y} = (0.076, 0.258, 0.369, 0.492, 0.559)$, using a least square estimator. To this purpose:

1. create two files, the main program `bod_fit.m` and the function `bod_ss.m` that computes the sum of squares objective function that is minimized. In the main program, we initialize the two parameters as `b_0 = [1 0.1]` and call the `fminsearch` optimizer as `[bmin,ssmin]=fminsearch(@bod_ss,b_0,[],data)`, while the function `bod_ss` must return `ss=bod_ss(theta,data)`.
2. To estimate the uncertainty of the parameters, we use the formula $Cov(\boldsymbol{\theta}) = \sigma^2(\mathbf{J}^\top\mathbf{J})^{-1}$, computing the Jacobian analytically. An estimate for the measurement error can be obtained using the mean square error
$$\sigma^2 \approx MSE = \frac{RSS}{n-p}$$
where RSS (residual sum of squares) is the fitted value of the least squares function, n is the number of measurements and p is the number of parameters.

1. The following code implements the least squares estimate:

```
%% LSQ fitting with the BOD model

clear all; close all; clc;
b_0 = [1 0.1];          % initial guess for the optimizer
x = (1:2:9)';           % x-data
y = [0.076 0.258 0.369 0.492 0.559]';       % y-data
data = [x,y];           % data matrix
n = length(x);          % number of data points

%% Get estimate for sigma**2 from the residual Sum of Squares
[bmin,ssmin]=fminsearch(@bod_ss,b_0,[],data);
```

The `fminsearch` optimizer takes in the sum of squares function, here defined in the separate file `bod_ss.m`. The second argument is the initial guess provided for the optimizer, and the third input contains the options structure with which one can control the optimizer (set tolerances, maximum number of iterations etc.). Here we use an empty matrix, which means that the default options are used. After the options structure, the user can pass some extra variables needed by the target function; here, we need the data matrix to compute the sum of squares.

The optimized function must return the target function value, and have the optimization variable as the first input argument, followed by the extra variables (listed after the options structure in the `fminsearch` call). That is, here we define the sum of squares in the form `ss=bod ss(theta,data)`:

```
function ss=bod_ss(theta,data)

x = data(:,1);
y = data(:,2);
ss = sum((y - theta(1)*(1-exp(-theta(2)*x))).^2);
end
```

2. We estimate the uncertainty of the parameters and compute the Jacobian matrix as follows:

```
 %% Compute the Jacobian analytically
J = [1-exp(-bmin(2).*x),x.*bmin(1).*exp(-x.*bmin(2))];

%% Compute the covariance and print the parameter estimates
sigma2 = ssmin/(n-2); % std of measurement noise estimated by the residuals
C = sigma2*inv(J'*J);
```

Note that computing the Jacobian can be easily done for this simple model, but usually a numerical approximation of the Jacobian is performed.

Finally, we print the LSQ estimates, standard deviations and t-values, and plot the model fit.

```
disp('(theta, std, t-values):');
[bmin(:) sqrt(diag(C)) bmin(:)./sqrt(diag(C))]

%% visualize the fit
xx=linspace(0,10);
yy=bmin(1)*(1-exp(-bmin(2)*xx));
plot(xx,yy,x,y,'ro');
xlabel('x'); ylabel('y=\theta_1(1-exp (\theta_2 x))');
```

We obtain, for the two parameters, the following estimates, standard deviations, and coefficients of variation:

```
                theta       std          coefficient of variation
theta1     0.9293     0.1195     7.7764
theta2     0.1040     0.0195     5.3306
```

The model fit is reported in Figure 1.

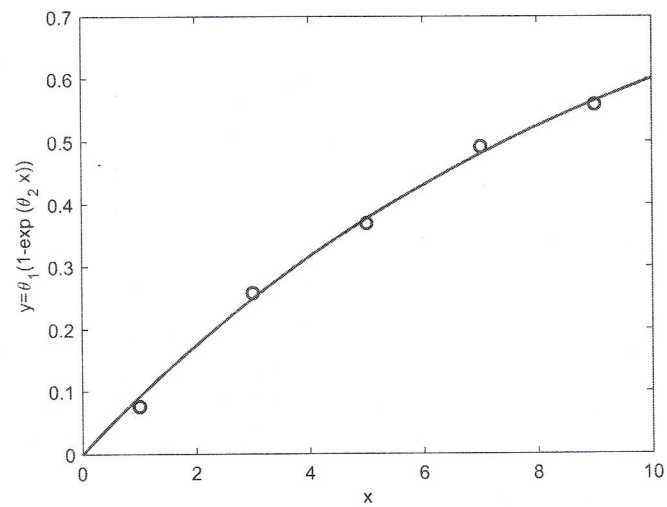


Figure 1: The model (blue line) fitted to the data (red circles).

(least squares fitting)

least squares for parameters estimation in a dynamical model

Exercise 2 (Parameter Estimation for Dynamical Models)
Let us consider the chemical reactions $A \to B \to C$, which can be modeled as an ODE system as follows:

$$\begin{aligned}
\frac{dA}{dt} &= -k_1 A, && t > 0 \\
\frac{dB}{dt} &= k_1 A - k_2 B, && t > 0 \\
\frac{dC}{dt} &= k_2 B, && t > 0 \\
A(0) &= 1 \\
B(0) &= 0 \\
C(0) &= 0
\end{aligned}$$

The unknown parameters are the reaction rate coefficients, $\boldsymbol{\theta} = (k_1, k_2)$. To estimate them, we want to use the least squares method, from the data on the compartments A and B:

we have 3 species that react together but we observe just 2 of them

```
% the data structure
data.time = 1:2:9;
data.ydata = [.504 .415
              .217 .594
              .101 .493
              .064 .394
              .008 .309];
```

Implement the model response calculation with a separate model function `ABCmodel.m` that solves the ODE system with the built-in solver `ode45`, and then determine the LS estimate, similarly to Exercise 1.

As in the previous example, we write the main program `ABCrun.m` that sets the data matrices and the auxiliary variables, performs the LSQ fitting using the `fminsearch` optimizer and graphs the fitting results. We define the data structure that is passed to the sum of squares target function and that contains the data and initial values for the ODE system, and call the optimizer:

```
% A --> B --> C demo
clear all; close all; clc;
% the data structure
data.time = 1:2:9;

data.ydata = [.504 .415
              .217 .594
              .101 .493
              .064 .394
              .008 .309];
data.y0 = [1 0 0];

% calling fminsearch
theta0=[1 1];
[thopt,ssopt]=fminsearch(@ABCss,theta0,[],data);
```

The sum of squares function `ABCss.m` computes the model response with given parameter values and compares it to the data:

```
function ss=ABCss(theta,data)

time=[0 data.time]; % adding zero to time vector!
ydata=data.ydata;
y0=data.y0;

[t,ymod]=ABCmodel(time,theta,y0);
ymod=ymod(2:end,1:2);  % taking A and B, removing initial value
ss=sum(sum((ydata-ymod).^2)); % the total SS
end
```

Note that for the ODE solver should start from the time point $t = 0$, and we therefore need to add the zero to the beginning of the time data vector, and then remove the corresponding row from the model response matrix `ymod`. Here, the model response calculation is done with a separate model function `ABCmodel.m` that solves the ODE system with the built-in solver `ode45`:

```
function [t,y]=ABCmodel(time,theta,y0)

[t,y]=ode45(@ABCode,time,y0,[],theta);
end
```

Finally, we need to implement the ODE function `ABCode.m` that specifies the ODE system for the solver. The function takes in the time point, model state and extra variables (in this order) and returns the derivatives of the system (as a column vector):

```
function dy=ABCode(t,y,theta)

% take parameters and components out from y and theta
k1=theta(1); k2=theta(2);
A=y(1); B=y(2);

% define the ODE
dy(1) = -k1*A;
dy(2) = k1*A-k2*B;
dy(3) = k2*B;
dy=dy(:);  % make sure that we return a column vector
```

Back in the main program `ABCrun.m`, after calling the optimizer, we compute the model response with the optimal parameter values, and visualize the model fit. The visualization code that produces Figure 2 reads as

```
t = linspace(0,10);
% visualization: solve model with thopt and compare to data t=linspace(0,10);
[t,ymod]=ABCmodel(t,thopt,data.y0);
plot(t,ymod);
hold on;
plot(data.time,data.ydata,'o');
hold off; xlabel('time'); ylabel('concentration');
legend('A','B','C');
```

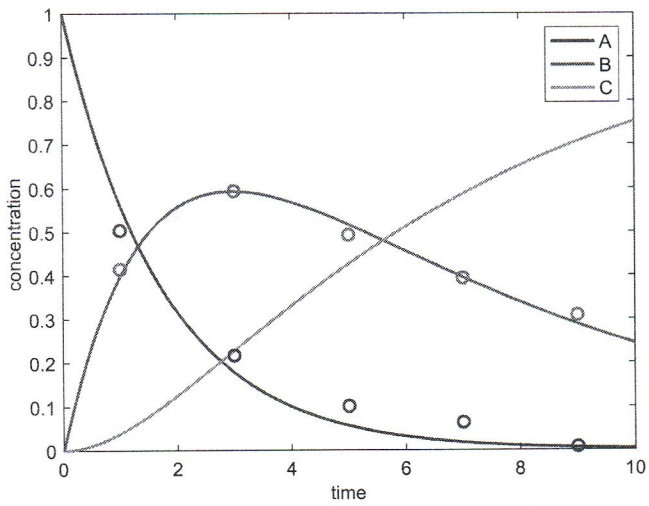


Figure 2: The model (lines) fitted to the data (circles).

Exercise 3 (Random Walk Metropolis algorithm).
Consider the BOD model of Exercise 1. The goal is to estimate the parameters and their uncertainty using the data $\mathbf{x} = (1, 3, 5, 7, 9)$ and $\mathbf{y} = (0.076, 0.258, 0.369, 0.492, 0.559)$, exploiting the Bayesian framework. To this purpose, we aim at implementing a MCMC procedure based on the Random Walk Metropolis algorithm, choosing as proposal distribution a bivariate Gaussian random variable with constant covariance matrix $\mathbf{C}$.

1. Implement the MCMC algorithm, setting to `nsimu=20000` the length of the chain, and to $C = 0.5 \cdot 10^{-3}\mathbf{I} \in \mathbb{R}^{2\times 2}$ the covariance matrix of the proposal distribution, computing also its Cholesky factor (we need to sample from a bivariate Gaussian...).
2. Display the acceptance rate, and the obtained chain.
3. Study how the parameter uncertainty affects the model predictions by simulating the model with different possible parameter values given by MCMC – the obtained trajectories provide what is often referred to as *predictive distribution*.
4. Check what happens to the computed chain when the covariance $\mathbf{C}$ of the Gaussian proposal distribution is evaluated analytically, about the LS estimate, as $\mathbf{C} = s_p\sigma^2(\mathbf{J}(\boldsymbol{\theta}_{LS})^\top\mathbf{J}(\boldsymbol{\theta}_{LS}))^{-1}$.

In particular, initialize the chain by considering the LS estimate (see Exercise 1 for the definition of the function `bod_ss.m`)

```
b_0 = [1 0.1];       % initial guess for the optimizer
x = (1:2:9)';        % x-data
y = [0.076 0.258 0.369 0.492 0.559]';    % y-data
data = [x,y];        % data matrix
n = length(x);       % number of data points

%% Get the LSQ estimate and estimate sigma**2 from the residuals
[bmin,ssmin]=fminsearch(@bod_ss,b_0,[],data);
sigma2 = ssmin/(n-2);
```

} we start from the least square estimator (that is a point in $\mathbb{R}^2$ since we have 2 parameters) and we'll use it to initialize the chain

For the initialization of the algorithm, consider the following commands:

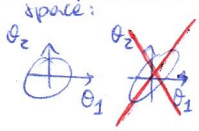
```
nsimu  = 20000;   # steps of the random walk metropolis
npar = 2;
chain  = zeros(nsimu,npar);

% spherical proposal covariance, try different scales!
qcov = 0.5e-3*eye(2);
R      = chol(qcov);        % the Cholesky 'square root' of qcov

%% initializations
oldpar      = bmin(:)';     % start the MCMC chain from the LSQ point
chain(1,:) = oldpar;
rej         = 0;            % n of rejected so far ...

%% Evaluate the SS at the starting point, and start the chain:
ss = bod_ss(oldpar,data);
SS=zeros(1,nsimu); SS(1) = ss;
```

no different scaling in different directions in the parameters space:



The `BOD_ss` function is implemented as follows:

```
function ss=bod_ss(theta,data)

x = data(:,1);
y = data(:,2);
ss = sum((y - theta(1)*(1-exp(-theta(2)*x))).^2);
end
```

1. We can implement the Random Walk Metropolis algorithm as follows:

```
%%%%% Metropolis MCMC algorithm
clear all; close all; clc;

b_0 = [1 0.1];      % initial quess for the optimizer
x = (1:2:9)';       % x-data
y = [0.076 0.258 0.369 0.492 0.559]';    % y-data
data = [x,y];       % data matrix
n = length(x);      % number of data points

%% Get the LSQ estimate and estimate sigma**2 from the residuals
[bmin,ssmin]=fminsearch(@bod_ss,b_0,[],data);
sigma2 = ssmin/(n-2);

%%%% Generate the MCMC chain
nsimu  = 20000;
npar = 2;
chain  = zeros(nsimu,npar);

% spherical proposal covariance, try different scales!
qcov = 0.5e-3*eye(2);

% Compute the Jacobian analytically
%J = [1-exp(-bmin(2).*x),x.*bmin(1).*exp(-x.*bmin(2))];
%scale = (2.4/sqrt(npar))^2;              % ?optimal? scaling
%qcov = scale*sigma2*inv(J'*J);           % proposal covariance

R      = chol(qcov);        % the Cholesky 'square root' of qcov

%% initializations
oldpar     = bmin(:)';      % start the MCMC chain from the LSQ point
chain(1,:) = oldpar;
rej        = 0;             % n of rejected so far ...

%% Evaluate the SS at the starting point, and start the chain:
ss = bod_ss(oldpar,data);
SS=zeros(1,nsimu); SS(1) = ss;

%% start the simulation loop
for i=2:nsimu
    newpar = oldpar+randn(1,npar)*R;
    ss2 = ss;
    ss1 = bod_ss(newpar,data);
    ratio = min(1,exp(-0.5*(ss1-ss2)/sigma2));
    if rand(1,1) < ratio
        chain(i,:) = newpar;        % accept
        oldpar     = newpar;
        ss         = ss1;
    else
        chain(i,:) = oldpar;        % reject
        rej = rej+1;
        ss = ss2;
    end
    SS(i) = ss;             % collect SS values
end
```

— implementation of the acceptance/rejection method

— sampling from a std gaussian, scale to the cholesky factor and add the old value (previous parameter)

2. We can display the acceptance rate and visualize the chain as follows:

```
% display the acceptance rate
disp('Acceptance rate');
accept = 1 - rej./nsimu

accept =
    0.1281

%% Visualization part

figure(1);
subplot(2,1,1);
plot(chain(:,1),'.');
ylabel('\theta_1');

subplot(2,1,2);
plot(chain(:,2),'.');
ylabel('\theta_2');
xlabel('MCMC step');

figure(2);
plot(chain(:,1),chain(:,2),'.');
xlabel('\theta_1'); ylabel('\theta_2'); axis tight;
```

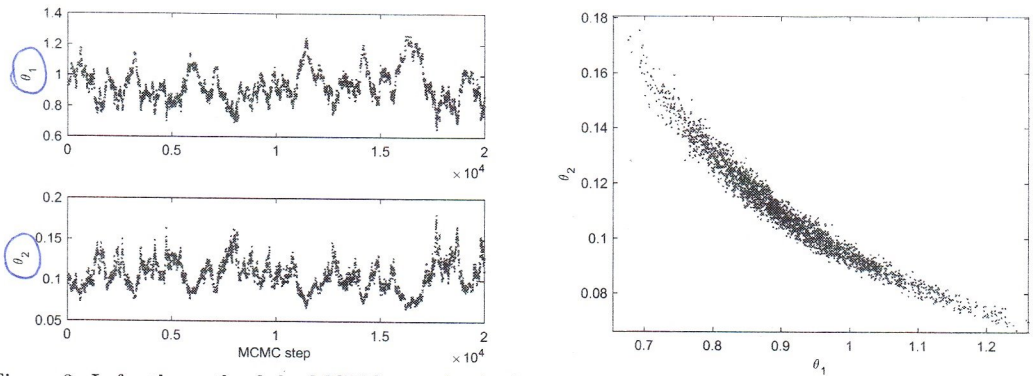


Figure 3: Left: the path of the MCMC sampler for both parameters. Right: the posterior distribution of the parameters.

The graphics produced by the commands above are reported in the Figure below. We can see that the MCMC method reveals a non-Gaussian, 'banana-shaped' posterior distribution, which is typical for nonlinear models. From the path of the sampler one can see that the mixing of the chain is not optimal. This is caused by the poorly chosen proposal covariance matrix: here we take a spherical covariance matrix, which ignores the correlations between the parameters visible in the posterior plots. A better choice would be a covariance matrix that is tilted so that it better matches the posterior distribution.

3. We can compute the predictive distribution as follows:

```
% predictive distribution
xx = linspace(0,40,50)';
c=1;
for i=1:10:nsimu
    ypred(:,c) = chain(i,1)*(1-exp(-chain(i,2)*xx));
    c=c+1;
end
yfit=bmin(1)*(1-exp(-bmin(2)*xx));
```

```
figure(3);
plot(xx,ypred,'r-',x,y,'bo', xx,yfit,'k-');
xlabel('x'); ylabel('y=\theta_1(1-exp (\theta_2 x))');
title('model prediction distribution & data');
```

to obtain the graph reported in Figure 4.

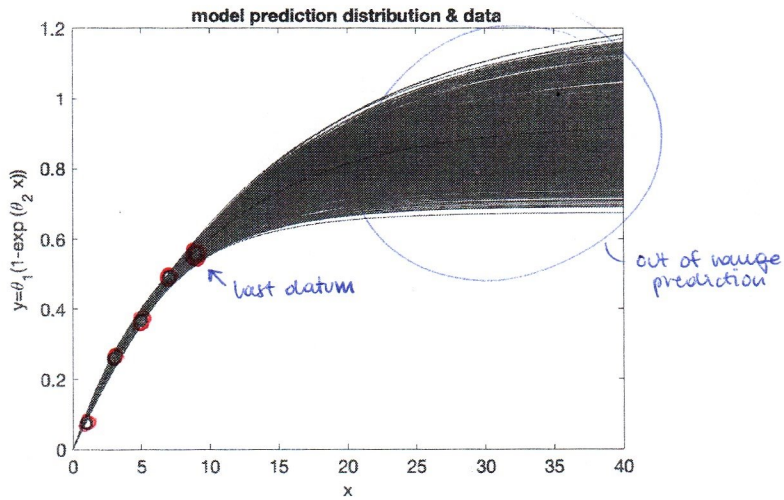


Figure 4: The predictive distribution of the model fit and prediction computed from the MCMC samples (red lines), the LSQ fit (black line) and the data points (blue circles).

4. We can replace the covariance matrix evaluating the Jacobian of the input-output map at the LS estimate as follows:

```
% Compute the Jacobian analytically
J = [1-exp(-bmin(2).*x),x.*bmin(1).*exp(-x.*bmin(2))];
scale = (2.4/sqrt(npar))^2; % "optimal" scaling
qcov = scale*sigma2*inv(J'*J); % proposal covariance
```

In this case the acceptance rates increases to 0.2532. The effect of the Jacobian-based proposal can be seen in the resulting plots, see the Figure above. The mixing of the chain is improved a lot, compared to the spherical proposal yielding the chain reported in Figure 5.

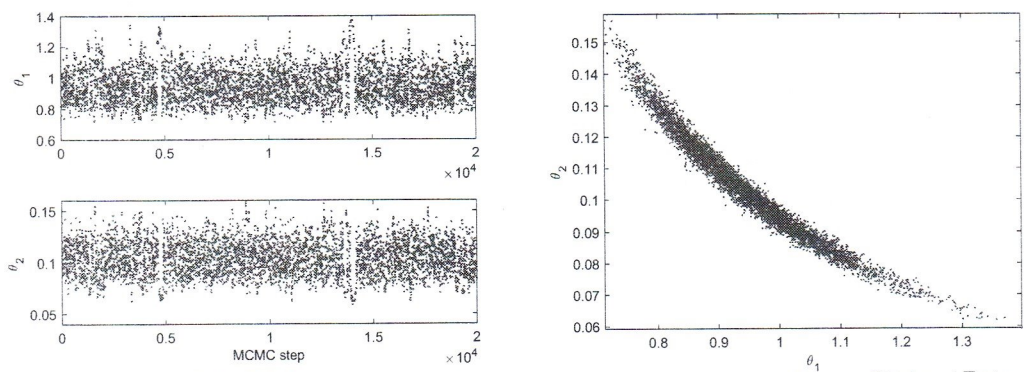


Figure 5: The path of the MCMC sampler for the two parameters with proposal $\mathbf{C} = s_p\sigma^2(\mathbf{J}(\boldsymbol{\theta}_{LS})^\top\mathbf{J}(\boldsymbol{\theta}_{LS}))^{-1}$ covariance matrix.

All the commands have been collected in the main program `BOD_MCMC_randomWalk_Metropolis.m`.

MCMC in practice: the mcmcstat Matlab package

We can take advantage of a Matlab package to make MCMC analyses easier to run. The code package can be freely downloaded from https://mjlaine.github.io/mcmcstat/. The toolbox provides a unified interface for specifying models, and implements, in addition to the basic Metropolis algorithm, the adaptive AM and DRAM methods. This is not the only available MCMC package in Matlab, however it is self-contained and quite easy to run. A recent adaptation in Python can be found at https://github.com/prmiles/pymcmcstat. In this section, we will demonstrate the use of the toolbox using the Example of **Exercise 3**, and the spring model introduced during lectures. Further examples can be found on the mcmcstat webpage.

mcmcstat provides tools to generate and analyze Metropolis-Hastings MCMC chains using multivariate Gaussian proposal distribution. The covariance matrix of the proposal distribution can be adapted during the simulation according to adaptive schemes. In particular, the code can:

- produce MCMC chain for user-written $-2 * \log(likelihood)$ and $-2 * \log(prior)$ functions. These will be equal to sum-of-squares cost functions when using Gaussian likelihood and prior.
- Do plots and statistical analyses based on the chain, such as basic statistics, convergence diagnostics, chain time series plots, 2 dimensional clouds of points, kernel densities, and histograms.
- Calculate densities, cumulative distributions, quantiles, and random variates for some useful common statistical distributions (without using Mathworks own statistics toolbox).

The code is self consistent, no additional Matlab toolboxes are used. The main functions in the toolbox are the following:

- mcmcrun.m. Matlab function for the MCMC run. The user provides her/his own Matlab function to calculate the "sum-of-squares" function for the likelihood part, e.g. a function that calculates minus twice the log likelihood, $-2\log(\pi(\mathbf{z} \mid \boldsymbol{\theta}))$. Optionally a prior "sum-of-squares" function can also be given, returning $-2\log(\pi_{prior}(\boldsymbol{\theta}))$.
- mcmcplot.m This function makes some useful plots of the generated chain, such as chain time series, 2 dimensional marginal plots, kernel density estimates, and histograms. See help mcmcplot.
- mcmcpred.m For certain types of models is is useful to plot predictive envelopes of model functions by sampling parameter values from the generated chain. This functions calls the model function repeatedly while sampling the unknowns from the chain. It calculates probability regions with respect to a given "time" variable of the model.

Example 1: the BOD model

Consider the problem of estimating $\boldsymbol{\theta} = (\theta_1, \theta_2)$ in the BOD model of **Exercise 3**, using the data $\mathbf{x} = (1, 3, 5, 7, 9)$ and $\mathbf{y} = (0.076, 0.258, 0.369, 0.492, 0.559)$. The main program for running the MCMC analysis is BOD_MCMC_mcmcstat_run.m. The code uses the same sum of squares function bod_ss.m as before.

To begin with, we set the data matrices and perform the LSQ fitting, as before, and (if not already done) add the mcmcstat folder that contains the toolbox files to the Matlab path:

```
%% MCMCRUN toolbox demo
clear all
% addpath mcmcstat;       % adding the mcmc package to the path
b_0 = [1 0.1];
x = (1:2:9)'; n = length(x);
y = [0.076 0.258 0.369 0.492 0.559]';
data = [x,y]; % observations

%% Get estimate for sigma2 from the residual Sum of Squares
[bmin,ssmin]=fminsearch(@bod_ss,b_0,[],data);
sigma2 = ssmin/(n-2);
```

Then, we start to specify the input structures needed by the `mcmcrun` function. The function needs four inputs: model, data, params and options. The model structure is used to define the sum of squares function and the measurement error variance. The sum of squares function must be implemented in the form that takes the parameter vector as the first argument and the data structure as the second argument and returns the sum of squares value: in our case, the function is defined as `ss=bod_ss(theta,data)`, as already done. In this case, the model structure is written as

```
%% the MCMC part
model.ssfun=@bod_ss; % SS function
model.sigma2=sigma2; % measurement error variance
```

Next, we define the `params` structure, that defines the parameter names, their starting values and possible minimum and maximum limits (in this order). The structure is given as a cell array, which in our case takes the following form:

```
% the parameter structure: name, init.val, min, max
params = {
    {'\theta_1',bmin(1),-Inf,Inf}
    {'\theta_2',bmin(2),-Inf,Inf} };
```

that is, we start the MCMC sampling from the LSQ estimate, and do not specify any bounds for the parameters.

Let us next specify the options structure, that controls how the MCMC sampler works. For instance, the number of samples, the (initial) proposal covariance matrix, the sampling method, and the adaptation interval for adaptive methods can be given via the options structure. In this case, we run the DRAM method for 20000 iterations, starting with a small proposal covariance $0.01\mathbf{I}$ and performing covariance adaptation at every 100th step:

```
% MCMC options
options.nsimu = 10000; % number of samples
options.qcov = 0.01*eye(2); % (initial) proposal covariance
options.method = 'dram'; % method: DRAM
options.adaptint = 100; % adaptation interval
```

At this point, we have everything we need to call the `mcmcrun` function. The function gives out a results structure and the sampled chain, and takes in the structures just defined:

```
% calling MCMCRUN
[results,chain] = mcmcrun(model,data,params,options);
```

The results can be visualized using the `mcmcplot` function included in the package, see `help mcmcplot`. Here we want two figures: one that draws the chain paths and one that plots the two-dimensional parameter distribution:

```
% visualizing the results using MCMCPLOT
figure(1);
mcmcplot(chain,[],results.names);

figure(2);
mcmcplot(chain,[],results.names,'pairs');
```

The code produces the plots reported in Figure 6. which are rather similar to the ones obtained earlier with the self coded Metropolis algorithm. Note that the adaptation has lead to a suitable proposal covariance matrix and the mixing of the chain is good. In addition to these, one can approximate the one dimensional marginal distributions, for instance, by histograms. In the `mcmcplot` function, one can give the format of the visualization as a parameter, `'chain'` gives the chain plot, `'pairs'` draws the two dimensional marginal distributions and `'hist'` gives histograms.

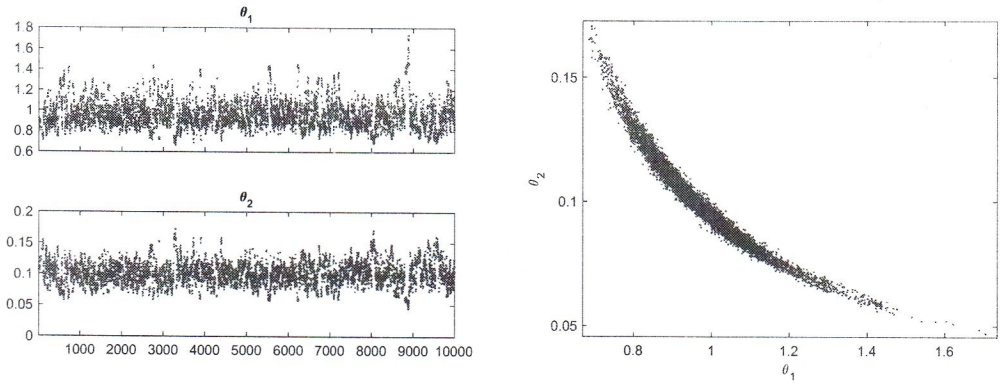


Figure 6: Left: the path of the MCMC sampler for both parameters. Right: the posterior distribution of the parameters.

The target density can be approximated based on the obtained samples also by the kernel density estimation technique. In kernel density estimation, the density is approximated by a sum of certain kernel functions, which are centered at the sampled parameter values. The kernel function can be, for instance, the density function of the normal distribution. The width and the orientation of the Gaussian kernel functions can be controlled via the covariance matrix. The wider the kernel is, the smoother density estimate we get, but a too wide kernel gives poor results, especially for the tails of the distribution. Kernel density estimation is implemented in the `mcmcplot` function. One can add density lines to the two dimensional marginal plots by using the function as

```
mcmcplot(chain,inds,names,'pairs',smo,rho)
```

where `smo` gives the width of the kernel and `rho` gives the orientation (correlation coefficient). If the orientation parameter is not given, a correlation coefficient computed from the MCMC samples is used.

In addition to visualizing the parameter posterior distribution, we are often interested in the predictive distribution, i.e., what is the uncertainty of the model predictions. Predictive distributions can be visualized simply by simulating the prediction model with different parameter values and drawing the prediction curves.

The MCMC package contains also functions for visualizing predictive distributions. The function `mcmcpred` simulates the model responses and computes different confidence envelopes for the predictions. The `mcmcpredplot` function can then be used to visualize the predictive distribution. In the BOD example, the following code can be used (see Figure 7):

```
bodmod=@(x,th) th(1)*(1-exp(-th(2)*x));      % bod model function
xx = linspace(0,40)';                        % x vector for plotting
out=mcmcpred(results,chain,[],xx,bodmod,500); % model predictions
mcmcpredplot(out);                           % the predictive distr.

hold on
plot(x,y,'bo','Linewidth',3);
xlabel('time'); ylabel('model response');
```

That is, the `mcmcpred` function takes in the results and chain variables produced by the `mcmcrun` function, the (optional) chain for the error variance (here empty matrix), the control variables with which the model is simulated (here the time vector), the model function, and the number of predictions computed.

Usually, one can simply visually inspect the chains to see how well the chain is mixing and if the sampler has reached its stationary distribution. However, various formal diagnostic methods have been developed to study if the sampler has converged. In the `mcmcstat` code package, the `chainstats` function can be used to compute some basic statistics of the chain. For the example at hand, the function prints the following table:

```
>> chainstats(chain,results)
                mean          std       MC_err          tau       geweke
----------------------------------------------------------------------
  \theta_1    0.97733      0.15254     0.013972       135.49      0.91688
  \theta_2     0.1008     0.019097    0.0011561       66.138      0.92174
----------------------------------------------------------------------
```

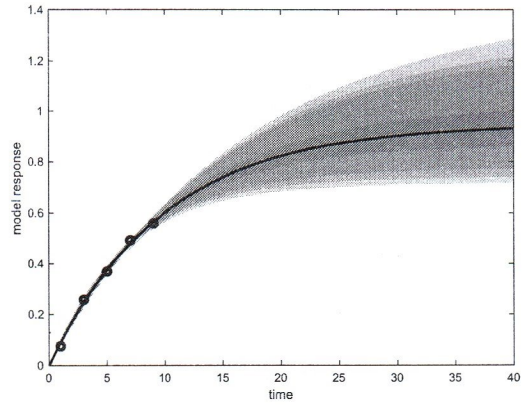


Figure 7: The data (blue dots), the median of the predictions (black line) and the 50%, 95% and 99% confidence envelopes for the predictions (grey areas).

showing the mean and the standard deviation of the chain. In addition we get an estimate of the MC standard error of the mean of the parameters, the integrated autocorrelation time and the Geweke convergence diagnostic[1].

A useful way to visualize how well the chain is mixing is to plot the autocorrelation function (ACF) of the parameter chains. The ACF tells how much, on average, samples that are k steps apart correlate with each other. In MCMC methods, subsequent points correlate with each other since the next point depends on the previous point. The further apart the samples are in the chain, the less they correlate. The ACF can be visualized with the `mcmcplot` using the plotting mode `'acf'`. In the BOD case, the ACF for steps $k = 1, \ldots, 100$ can be plotted as follows:

```
mcmcplot(chain,[],results.names,'acf',100);
```

The code produces the autocorrelation plots for both parameters, given in Figure 8. One can see that the autocorrelation goes towards zero, and reaches the zero level at around $k = 80$. This means roughly that taking every $80th$ member of the chain results in uncorrelated samples. The ACF plot is often used to compare the efficiency of MCMC schemes: the faster the autocorrelation goes to zero, the better.

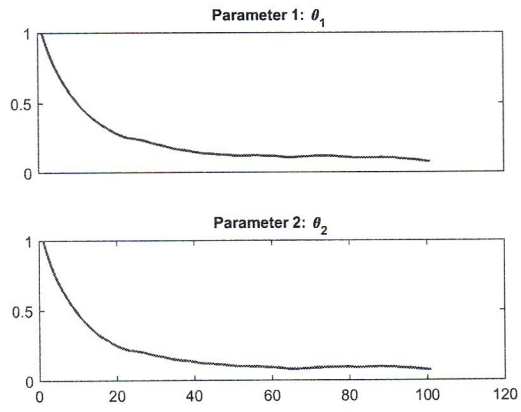
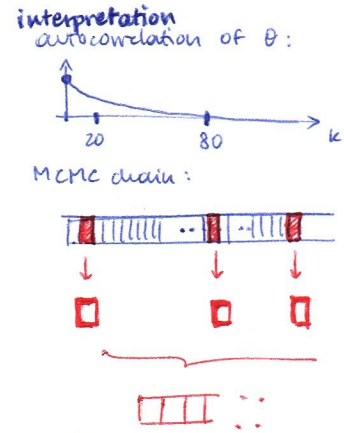


Figure 8: The autocorrelation function for two parameters..



if we take one sample every 20 then we obtain a collection which is not independent (since autocorrelation at k=20 is not 0)

if, instead, we consider one sample every 80 then the resulting collection will be independent (= samples will be independent

because of this, we want autocorrelation to go to zero as fast as possible

[1]The Geweke diagnostic is a univariate diagnostic that is usually applied to each marginal posterior distribution. This diagnostic is based on a test for equality of the means of the first and last part of a Markov chain (by default the first 10% and the last 50%). If the samples are drawn from a stationary distribution of the chain, then the two means are equal and Geweke's statistic has an asymptotically standard normal distribution. The idea is to mimic the simple two-sample test of means: if the mean of the first 10% is not significantly different from the last 50%, then we conclude the target distribution converged somewhere in the first 10% of the chain. If the p-values from the Z tests are greater than 0.05, we may treat the first 10% as burn-in. The p-value of the test about the two-sample test of means is reported.

Example 2: the spring model

Consider the spring model (already introduced during lectures)

$$\begin{cases} u'' + Cu' + Ku = 0, & t > 0 \\ u(0) = 2 \\ u'(0) = -C \end{cases}$$

with displacement observation, so that

$$z = \begin{pmatrix} 1 & 0 \end{pmatrix} \begin{pmatrix} u \\ u' \end{pmatrix} = u.$$

The previous problem admits the solution

$$u(t) = 2e^{-Ct/2} \cos(\sqrt{K - C^2/4}\, t)$$

provided that $C^2 - 4K < 0$. We take $K = 20.5$ to be known and let $\theta = C$ be the parameter considered in the statistical analysis. Note that the dependence of $u(t, \theta)$ on θ is nonlinear.

To numerically generate synthetic data, we employ $C_0 = 1.5$ and add noise $\varepsilon \sim N(0, \sigma_0^2)$ where $\sigma_0 = 0.1$.

Exercise 4 (Spring model)
We want to use `mcmcstat` to identify the parameter $\theta = C$ in the spring model. To this purpose:

1. Set up a function `function y = spring_fun(t,params)` that returns, as a row vector, the (here, exact) solution of the ODE above, evaluated at the vector `t`.
2. Set up a function `function ss = spring_ss(params,data)` that returns the sum of squares function as a function of the parameter C and the data.
3. Use the `mcmcrun` function to generate the chain, and perform MCMC diagnostic. Collect all the commands in the script `spring_dram`.
4. Construct and plot prediction intervals using the `mcmcpred` function (and, possibly, compare the obtained results with the intervals constructed by hands).

Use as initialization of the `mcmcrun` function the following commands, at the beginning of the script `spring_dram`:

```
global m k

%% Input the parameters and construct synthetic data
m = 1;  c = 1.5;  k = 20.5;
sigma = .1;  var = sigma^2;

num = sqrt(4*m*k - c^2);
den = 2*m;

t = 0:.01:5;
error = sigma*randn(size(t));
n = length(t);

y = 2*exp((-c/den)*t).*cos((num/den)*t);
obs = y + error;

cmin = 1.4792;  % LSQ estimate

data.xdata = t;
data.ydata = obs;
```