Statistical Analysis in Modeling

Antti Solonen*, Heikki Haario*, Marko Laine†

∗ Lappeenranta University of Technology

† Finnish Meteorological Institute

Contents

1 Introduction

This material is prepared for the LUT course *Statistical Analysis in Modeling*. The purpose of the course is to give a practical introduction into how statistics can be used to quantify uncertainties in mathematical models. The goal is that after the course the student is able to use modern computational tools, especially different Monte Carlo and Markov Chain Monte Carlo (MCMC) methods, for estimating the reliability of modeling results. The material should be accessible with knowledge of basic engineering mathematics (especially matrix calculus) and basics of statistics, and the course is therefore suitable for engineering students from different fields.

The course contains a lot of practical numerical exercises, and the software used in the course is MATLAB. To successfully complete the course, the student should have basic skills in MATLAB or another platform for numerical computations.

In the course, we will use the MCMC code package written by Marko Laine. Download the code from `http://helios.fmi.fi/~lainema/mcmc/mcmcstat.zip`. Some additional codes that are needed to follow the examples of this document are provided in the website of the course, download packages `utils.zip` and `demos.zip`.

1.1 Mathematical Models

Mathematical modeling is a central tool in most fields of science and engineering. Mathematical models can be either 'mechanistic', which are based on principles of natural sciences, or 'empirical' where the phenomenon cannot be modeled exactly and the model is built by inferring relationships between variables directly from the available data.

Empirical models, also often called 'soft' models or 'data-driven' models, are based merely on existing data, and do not involve equations derived using principles of natural sciences. That is, we have input variables $\mathbf{X}$ and output variables $\mathbf{Y}$, and we wish to learn the dependency between $\mathbf{X}$ and $\mathbf{Y}$ using the empirical data alone. If the input values consist of x–variables $\mathbf{x}_1, ..., \mathbf{x}_p$, the output of y–variables $\mathbf{y}_1, ..., \mathbf{y}_q$ and we have n experimental measurements done, the data values may be expressed as a *design* and *response* matrices,

$$\mathbf{X} = \begin{array}{c} \begin{matrix} \mathbf{x}_1 & \mathbf{x}_2 & \dots & \mathbf{x}_p \end{matrix} \\ \begin{pmatrix} x_{11} & x_{12} & \dots & x_{1p} \\ x_{21} & x_{22} & \dots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \dots & x_{np} \end{pmatrix} \end{array} \quad \mathbf{Y} = \begin{array}{c} \begin{matrix} \mathbf{y}_1 & \mathbf{y}_2 & \dots & \mathbf{y}_q \end{matrix} \\ \begin{pmatrix} y_{11} & y_{12} & \dots & y_{1q} \\ y_{21} & y_{22} & \dots & y_{2q} \\ \vdots & \vdots & \ddots & \vdots \\ y_{n1} & y_{n2} & \dots & y_{nq} \end{pmatrix} \end{array}. \tag{1}$$

Empirical models often result in easier computation, but there can be problems in interpreting the modeling results. The main limitation of the approach is that the models can be poorly extended to new situations from which measurements are not available (extrapolation). An empirical model is often the preferred choice, if a 'local' description (interpolation) of the phenomenon is enough or if the mechanism of the phenomenon is not known or too complicated to be modeled in detail. The methodology used in analyzing empirical model is often called *regression analysis*.

Mechanistic models, also known as 'hard' models or 'physio-chemical' models, are built on first principles of natural sciences, and are often formulated as differential equations. For instance, let us consider an elementary chemical reaction $A \to B \to C$. The system can be modeled by mass balances, which leads to an ordinary differential equation (ODE) system

$$\begin{aligned}\frac{dA}{dt} &= -k_1 A \\ \frac{dB}{dt} &= k_1 A - k_2 B \\ \frac{dC}{dt} &= k_2 B.\end{aligned}$$

The model building is often more demanding as with empirical models, and also require knowledge of numerical methods, for instance, for solving the differential equations. In the example above, we can employ a standard ODE solver for solving the model equations. For more complex models, for instance computational fluid dynamics (CFD) models require various more advanced numerical solution methods, such as Finite Difference (FD) or finite element (FEM) methods.

The benefit of the physio-chemical approach is that it often extends well to new situations. The mechanistic modeling approach can be chosen if the phenomenon can understood well enough and if the model is needed outside the region where the experiments are done.

In a typical modeling project, a mathematical model is formulated first and data is then collected to verify that the model is able to describe the phenomenon of interest. The models usually contain some unknown quantities (parameters) that need to be estimated from the measurements; the goal is to set the parameters so that the model explains the collected data well. In this course, we will use the following notation:

$$\mathbf{y} = f(\mathbf{x}, \theta) + \varepsilon, \tag{2}$$

where $\mathbf{y}$ are the obtained measurements and $f(\mathbf{x}, \theta)$ is the mathematical model that relates certain control variables $\mathbf{x}$ and unknown parameters θ to the measurements. The measurement error is denoted by ε.

Let us clarify the notation using the chemical reaction model given above. The unknown parameters θ are the reaction rate constants (k_1, k_2). The design variables $\mathbf{x}$ in such a model can be measurement time instances and temperatures, for instance. The measurement vector $\mathbf{y}$ can contain, e.g., the measured concentration of some of the compounds in the reaction. The model f is the solution of the corresponding components of the ODE computed at measurement points $\mathbf{x}$ with parameter values θ.

Mathematical models are either linear or nonlinear. Here, we mean linearity with respect to the unknown parameters θ. That is, for instance $y = \theta_0 + \theta_1 x_1 + \theta_2 x_2$ is linear, whereas $y = \theta_1 \exp(\theta_2 x)$ is nonlinear. Most of the classical theory is for linear models, and linear models often result in easier computations.

1.2 Uncertainties in Mathematical Models

Models are simplifications of reality, and therefore model predictions are always uncertain. Moreover, the obtained measurements, with which the models are calibrated, are often noisy. The task of statistical analysis of mathematical models is to quantify the uncertainty in the models. The statistical analysis of the model happens at the 'model fitting' stage: uncertainty in the data $\mathbf{y}$ implies uncertainty in the parameter values θ.

Traditionally, point estimates for the parameters are obtained, for example, by solving a least squares optimization problem. In the least squares approach, we search for parameter values $\hat{\theta}$ that minimize the sum of squared differences between the observations and the model:

$$SS(\theta) = \sum_{i=1}^{n} [y_i - f(x_i, \theta)]^2. \tag{3}$$

That is, the least squares estimator is $\hat{\theta} = \text{argmin } SS(\theta)$. The least squares method as such does not say anything about the uncertainty related to the estimator, which this is the purpose of statistical analysis in modeling.

The theory for statistical analysis of linear models is well established, and traditionally the linear theory is used as an approximation for nonlinear models as well. However, efficient numerical Monte Carlo techniques, such as Markov Chain Monte Carlo (MCMC), have been introduced lately to allow full statistical analysis for nonlinear models. The main goal of this course is to introduce these methods to the student in a practical manner.

This course starts by reviewing the classical techniques for statistical analysis of linear and nonlinear models. Then, the Bayesian framework for parameter estimation is presented, together with MCMC methods for performing the numerical computations. Bayesian estimation and MCMC can be considered as the main topic of this course.

2 Prequisities

In this section, we briefly recall the basic concepts in statistics that are needed in this course. In addition to the statistics topics given below, the reader should be familiar with basic concepts in matrix algebra.

2.1 Statistical Measures

In this course, we will need the basic statistical measures: *expectation*, *variance*, *covariance* and *correlation*. The expected value for a random vector $\mathbf{x}$ with probability density function (PDF) $p(\mathbf{x})$ is

$$\mathrm{E}(\mathbf{x}) = \int \mathbf{x} p(\mathbf{x}) dx, \tag{4}$$

where the integral is computed over all possible values of $\mathbf{x}$, in this course typically of the d-dimensional Euclidean space $\mathbb{R}^d$. The variance of a one-dimensional random variable x, which measures the amount of variation in the values of x, is defined as the expected squared deviation of x from its expected value:

$$\mathrm{Var}(x) = \int (x - \mathrm{E}(x))^2 p(x) dx. \tag{5}$$

The *standard deviation* is defined as the square root of the variance, $\mathrm{Std}(x) = \sqrt{\mathrm{Var}(x)}$.

The *covariance* of two one-dimensional random variables x and y measures how much the two variables change together. Covariance is defined as

$$\mathrm{Cov}(x, y) = \mathrm{E}((x - \mathrm{E}(x))(y - E(y))). \tag{6}$$

The *covariance matrix* of two random vectors $\mathbf{x}$ and $\mathbf{y}$, with dimensions n and m, gives the covariances of different combinations of elements in the $\mathbf{x}$ and $\mathbf{y}$ vectors. The covariances are collected into an $n \times m$ covariance matrix, defined as

$$\mathrm{Cov}(\mathbf{x}, \mathbf{y}) = \mathrm{E}((\mathbf{x} - \mathrm{E}(\mathbf{x}))(\mathbf{y} - E(\mathbf{y}))^T). \tag{7}$$

In this course, the covariance matrix is used to measure the covariances within the elements of a single random vector $\mathbf{x}$, and we denote the covariance matrix by $\mathrm{Cov}(\mathbf{x})$. The covariance matrix is simply defined as the covariance matrix between $\mathbf{x}$ and itself: $\mathrm{Cov}(\mathbf{x}) = \mathrm{Cov}(\mathbf{x}, \mathbf{x})$. The diagonals of the $n \times n$ covariance matrix contain the variances of individual elements of the vector, and off-diagonals give the covariances between the elements.

The correlation coefficient between two random variables x and y is the normalized version of the covariance, and is defined as

$$\mathrm{Cor}(x, y) = \frac{\mathrm{Cov}(x, y)}{\mathrm{Std}(x)\mathrm{Std}(y)}. \tag{8}$$

For vector quantities, one can form a *correlation matrix* (similarly as in the covariance matrix), that has ones in the diagonal (correlation of elements with themselves) and correlation of separate elements in the off-diagonal.

2.2 Sample Statistics

To be able to use the statistical measures defined above, we need ways to estimate the statistics using measured data. Here, we give the *sample statistics*, which are *unbiased* estimators of the above measures in the sense that they approach the true values of the measures as the number of measurements increases.

Let us consider a $n \times p$ matrix of data, where each column contains n measurements for variables $\mathbf{x}_1, ..., \mathbf{x}_p$:

$$\mathbf{X} = \begin{pmatrix} \mathbf{x}_1 & \mathbf{x}_2 & \dots & \mathbf{x}_p \\ x_{11} & x_{12} & \dots & x_{1p} \\ x_{21} & x_{22} & \dots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \dots & x_{np} \end{pmatrix}. \tag{9}$$

In this course, we need the following sample statistics:

- the sample mean for the kth variable is

$$\hat{x}_k = \frac{1}{n} \sum_{i=1}^{n} x_{ik} \tag{10}$$

- the sample variance of the kth variable is

$$\mathrm{Var}(x_k) = \frac{1}{n-1} \sum_{i=1}^{n} (x_{ik} - \hat{\mathbf{x}}_k)^2 = \sigma_k^2 \tag{11}$$

- the sample standard deviation is

$$\mathrm{Std}(x_k) = \sqrt{(\mathrm{Var}(x_k))} = \sigma_k \tag{12}$$

- the sample covariance between two variables is

$$\mathrm{Cov}(x_k, x_l) = \frac{1}{n-1} \sum_{i=1}^{n} (x_{ik} - \bar{x}_k)(x_{il} - \bar{x}_l) := \sigma_{x_k x_l} \tag{13}$$

- the sample correlation coefficient between two variables is

$$\mathrm{Cor}(x_k, x_l) = \frac{\sigma_{x_k x_l}}{\sigma_{x_k} \sigma_{x_l}} := \rho_{x_k x_l} \tag{14}$$

The sample variances and covariances for p variables can be compactly represented as a sample covariance matrix, which is given as

$$\mathrm{Cov}(\mathbf{X}) = \begin{bmatrix} \sigma_{x_1}^2 & \sigma_{x_1 x_2} & \dots & \sigma_{x_1 x_p} \\ \sigma_{x_1 x_2} & \sigma_{x_2}^2 & & \\ \vdots & & \ddots & \\ \sigma_{x_1 x_p} & & & \sigma_{x_p}^2 \end{bmatrix}. \tag{15}$$

That is, the variances of the variables are given in the diagonal of the covariance matrix, and the off-diagonal elements give the individual covariances. Similarly, one can form a correlation matrix using the relationship of covariance and correlation given in equation (14), with ones in the diagonal and correlation coefficients between the variables in the off-diagonal:

$$\mathrm{Cor}(\mathbf{X}) = \begin{bmatrix} 1 & \rho_{x_1x_2} & \cdots & \rho_{x_1x_p} \\ \rho_{x_1x_2} & 1 & & \\ \vdots & & \ddots & \\ \rho_{x_1x_p} & & & 1 \end{bmatrix}. \tag{16}$$

Note that here we use partly the same notation for the definition of the statistical measures and their corresponding sample statistics. In the rest of the course, we will only need the sample statistics, and in what follows, *Var* means sample variance, *Cov* means sample covariance matrix and so on.

2.3 Some distributions

In this course, we need some basic distributions. The distribution of a random variable $\mathbf{x}$ is characterized by its probability density function (PDF) $p(\mathbf{x})$.

The PDF of the univariate normal (or Gaussian) distribution with mean x_0 and variance σ^2 is given by the formula

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{1}{2}\left(\frac{x - x_0}{\sigma}\right)^2\right). \tag{17}$$

For a random variable x that follows the normal distribution, we write $x \sim \mathrm{N}(x_0, \sigma^2)$.

Let us consider a sum of n univariate random variables, $s = \sum_{i=1}^{n} x_i^2$. The sum follows the chi-square distribution with n degrees of freedom, which we denote by $s \sim \chi_n^2$. The density function for the chi-square distribution is

$$p(s) = 1/(2^{n/2}\Gamma(n/2))s^{n/2-1}e^{-s/2}. \tag{18}$$

The most important distribution in this course is the multivariate normal distribution. The d-dimensional Gaussian distribution is characterized by the $d \times 1$ mean vector $\mathbf{x}_0$ and the $d \times d$ covariance matrix Σ. The probability density function is

$$p(\mathbf{x}) = \frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x} - \mathbf{x}_0)^T\Sigma^{-1}(\mathbf{x} - \mathbf{x}_0)\right), \tag{19}$$

where $|\Sigma|$ denotes the determinant of Σ.

Let us consider a zero-mean multivariate Gaussian with covariance matrix Σ. The term in the exponential can be written as $\mathbf{x}^T\Sigma^{-1}\mathbf{x} = \mathbf{y}^T\mathbf{y}$, where $\mathbf{y} = \Sigma^{-1/2}\mathbf{x}$. Here $\Sigma^{1/2}$ and $\Sigma^{-1/2}$ denote the 'square roots' (e.g. Cholesky decompositions) of the covariance matrix and its inverse, respectively. Using the formula for computing

the covariance matrix of a linearly transformed variable, see equation (22) in the next section, the covariance of the transformed variable can be written as

$$\mathrm{cov}(\Sigma^{-1/2}\mathbf{x}) = \Sigma^{-1/2}\mathrm{cov}(\mathbf{x})(\Sigma^{-1/2})^T = \Sigma^{-1/2}\Sigma(\Sigma^{-1/2})^T = \Sigma^{-1/2}\Sigma^{1/2}(\Sigma^{1/2}\Sigma^{-1/2})^T = \mathbf{I}, \tag{20}$$

where $\mathbf{I}$ is the identity matrix. That is, the term $\mathbf{x}^T\Sigma^{-1}\mathbf{x} = \mathbf{y}^T\mathbf{y}$ is a squared sum of d independent standard normal variables, and therefore follows the chi-square distribution with d degrees of freedom:

$$\mathbf{x}^T\Sigma^{-1}\mathbf{x} \sim \chi^2_d. \tag{21}$$

This gives us a way to check how well a sampled set of vectors follows a Gaussian distribution with a known covariance matrix: compute the values $\mathbf{x}^T\Sigma^{-1}\mathbf{x}$ for each sample and see how well the statistics of the values follow the χ^2_d distribution (see exercises).

The contours of multivariate Gaussian densities are ellipsoids, where the principal axes are given by the eigenvectors of the covariance matrix, and the eigenvalues correspond to the lengths of the axes. Therefore, two-dimensional covariance matrices can be visualized as ellipses. In practice, this can be done with the `ellipse` function given in the code package, see `help ellipse`. In Fig. 2.3 below, ellipses corresponding to three different covariance matrices are illustrated (the figure is produced by the program `ellipse_demo.m`).

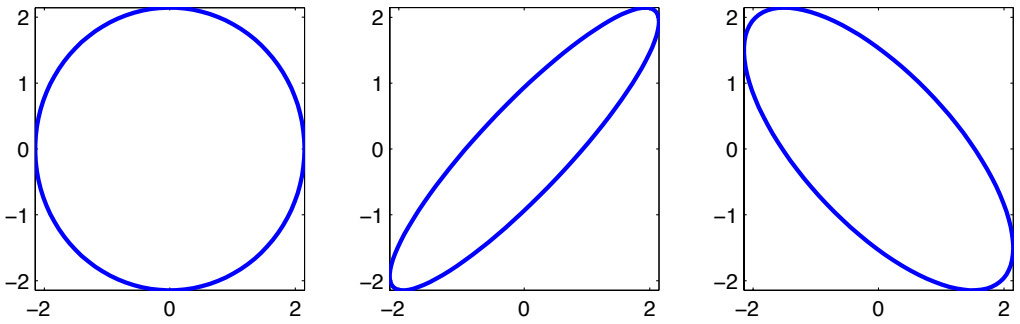

Figure 1: Ellipses corresponding to the 90% confidence region for three covariance matrices, diagonal (left), with correlation 0.9 (center) and with correlation −0.7 (right).

2.4 Generating Gaussian random numbers

Later in the course, we need to be able to generate random numbers from multivariate Gaussian distributions. A basic strategy is given here. Recall that in the univariate case, normal random numbers $y \sim N(\mu, \sigma^2)$ are generated by $y = \mu + \sigma z$, where $z \sim N(0, 1)$.

It can be shown (see exercises) that the covariance matrix of a random variable multiplied by a matrix $\mathbf{A}$ is

$$\mathrm{cov}(\mathbf{Ax}) = \mathbf{A}\mathrm{cov}(\mathbf{x})\mathbf{A}^T. \tag{22}$$

This is a useful formula when random variables are manipulated. For instance, let us consider a covariance matrix Σ, which we can decompose into a 'square-root' form $\Sigma = \mathbf{A}\mathbf{A}^T$. Now, if we take a random variable with $\mathrm{cov}(\mathbf{x}) = \mathbf{I}$, the covariance matrix of the transformed variable $\mathbf{A}\mathbf{x}$ is

$$\mathrm{cov}(\mathbf{A}\mathbf{x}) = \mathbf{A}\mathrm{cov}(\mathbf{x})\mathbf{A}^T = \mathbf{A}\mathbf{A}^T = \Sigma. \tag{23}$$

This gives us a recipe for generating Gaussian random variables with a given covariance matrix:

ALGORITHM: generating zero mean Gaussian random variables y with covariance matrix Σ

1. Generate standard normal random numbers $\mathbf{x} \sim \mathrm{N}(\mathbf{0}, \mathbf{I})$.
2. Compute decomposition $\Sigma = \mathbf{A}\mathbf{A}^T$.
3. Transform the standard normal numbers with $\mathbf{y} = \mathbf{A}\mathbf{x}$.

In practice, the standard normal random numbers in step 1 can be generated using existing routines built in to programming environments (in MATLAB using the `randn` command). The matrix square root needed in step 2 can be computed, for instance, using the Cholesky decomposition (the `chol` function in MATLAB). However, note that any symmetric decomposition $\Sigma = \mathbf{A}\mathbf{A}^T$ will do. Note that the approach is analogical to the one-dimensional case, where standard normal numbers are multiplied with the square root of the variance (standard deviation). Non-zero Gaussian variables can simply be created by adding the mean vector to the generated samples. Generating multivariate normal random vectors is demonstrated in the program `randn_demo.m`

Gaussian random number generation has also an intuitive geometric interpretation. If the covariance matrix can be written as a sum of outer products of n vectors $(a_1, ..., a_n)$,

$$\Sigma = \sum_{i=1}^{n} \mathbf{a}_i \mathbf{a}_i^T, \tag{24}$$

it is easy to verify that the sum

$$\mathbf{y} = \sum_{i=1}^{n} \omega_i \mathbf{a}_i, \tag{25}$$

where ω_i are standard normal scalars, $\omega_i \sim N(0, 1)$, follows the Gaussian distribution with covariance matrix Σ. That is, Gaussian random variables can be viewed as randomly weighted combinations of 'basis vectors' $(\mathbf{a}_1, ..., \mathbf{a}_n)$. This interpretation is illustrated for a two-dimensional case in Fig. 2.4. For an animation, see also the demo program `mvnrnd_demo.m`.

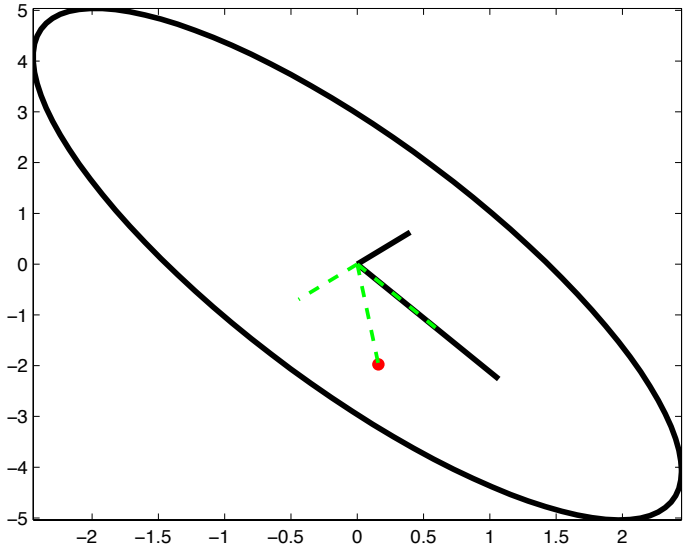

Figure 2: Black lines: two vectors $\mathbf{a}_1$ and $\mathbf{a}_2$ and the 95% confidence ellipse corresponding to the covariance matrix $\Sigma = \sum_{i=1}^{2} \mathbf{a}_i \mathbf{a}_i^T$. Green lines: weighted vectors $\omega_1 \mathbf{a}_1$ and $\omega_2 \mathbf{a}_2$. Red dot: one sampled point $\mathbf{y} = \sum_{i=1}^{2} \omega_i \mathbf{a}_i$.

2.5 General Methods for Generating Random Numbers

In this section, we consider two methods to sample from non-standard distributions, whose density function is known: the inverse CDF technique and the accept-reject method. Another important class of random sampling methods, the Markov chain Monte Carlo (MCMC) methods, is covered separately in Section 6.

Inverse CDF Method

Let us consider a continuous random variable whose cumulative distribution function (CDF) is F. The inverse CDF method is based on the fact that a random variable $x = F^{-1}(u)$ where u is sampled from $U[0, 1]$, and F^{-1} is the inverse function of F, has the distribution with CDF F. The inverse CDF method can be visualized so that uniform random numbers are 'shot' from the y-axis to the CDF curve and the corresponding points in the x-axis are samples from the correct target, as illustrated for the exponential distribution in Fig. 2.5.

The inverse CDF method is applicable, if the cumulative distribution function is invertible. For distributions with non-invertible CDF, other methods need to be applied. If one has a lot of samples, one can approximate the CDF of the distribution by computing the empirical CDF function, and generate random variables using that. In addition, the inverse CDF method is a good way to draw samples from discrete distributions, where the CDF is essentially a step function.

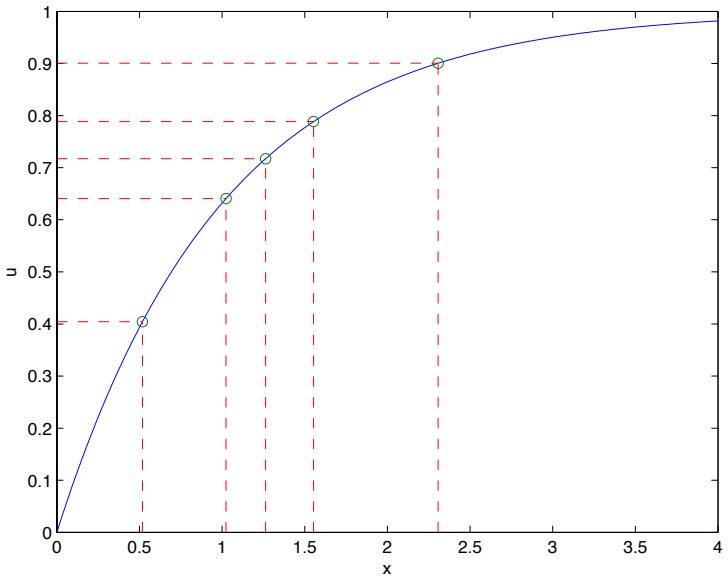

Figure 3: Producing samples from the exponential distribution using inverse CDF, $F(x) = 1 - \exp(-x)$ and $F^{-1}(u) = -\ln(1-u)$.

Accept-Reject Method

Suppose that f is a positive (but non-normalized) function on the interval $[a, b]$, bounded by M. If we generate a uniform sample of points (x_i, u_i) on the box $[a, b] \times [0, M]$, the points that satisfy $u_i < f(x_i)$ form a uniform sample under the graph of f.

The area of any slice $\{(x, y)|x_l < x < x_u, y \leq f(x)\}$ is proportional to the number of sampled points in it. So the (normalized) histogram of the points x_i gives an approximation of the (normalized) PDF given by f. The accept-reject method is illustrated in Fig. 2.5

The method is not restricted to 1D, the interval $[a, b]$ may have any dimension. We arrive at the following algorithm:

- Sample $x \sim U([a, b])$, $u \sim U([0, M])$.
- Accept points x for which $u < f(x)$

So, we have a (very!) straightforward method that, in principle, solves 'all' sampling problems. However, difficulties arise in practice: how to choose M and the interval $[a, b]$. M must be larger that the maximum of f, but too large a value leads to many rejected samples. The interval $[a, b]$ should contain the support of f – which generally is unknown. A too large interval again leads to many rejections. Moreover, uniform sampling in a high dimensional interval is inefficient in any case, if the support of f only is a small subset of it.

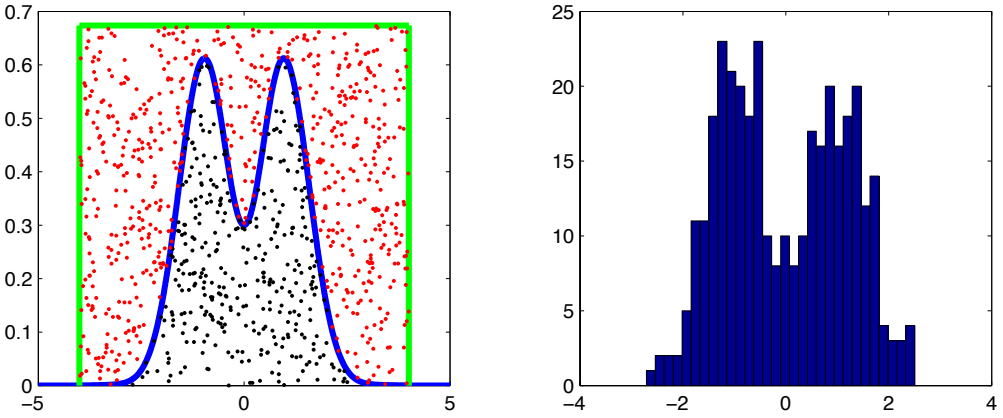

Figure 4: Left: In accept-reject, points are sampled in the box $[a, b] \times [0, M]$ (green line), and points that fall below the target density function (blue line) are accepted (black points) and the other points are rejected (red points). Right: the histogram of the sampled points.

2.6 Exercises

1. Generate random numbers from $\mathrm{N}(\mu, \sigma^2)$ with $\mu = 10$ and $\sigma = 3$. Study how the accuracy of the empirical mean and standard deviation behaves as the sample size increases.

2. Generate samples from the χ^2_n distribution with $n = 2$ and $n = 20$ in three different ways:

   (a) by the definition (computing sums of normal random numbers)

   (b) by the inverse CDF method (e.g. using the `chi2inv` MATLAB function)

   (c) directly by the `chi2rnd` MATLAB function

   For each case, create an empirical density function (normalized histogram) and compare it with the true density function (obtained e.g. by the `chi2pdf` MATLAB function).

3. Generate random samples from a zero-centered Gaussian distribution with covariance matrix

$$\mathbf{C} = \begin{pmatrix} 2 & 5 \\ 5 & 16 \end{pmatrix}.$$

   Verify by simulation that approximately a correct amount of the sampled points are inside the 95% and 99% confidence regions, given by the limits of the χ^2 distribution (use the `chi2inv` function in MATLAB). Visualize the samples and the confidence regions as ellipses (use the `ellipse` function).

4. Show that equation (22) holds.

5. Using the Accept–Reject method, generate samples from the (non-normalized) density

$$f(x) = \mathrm{e}^{-x^2/2}(\sin(6x)^2 + 3\cos(x)^2 \sin(4x)^2 + 1)$$

As the dominating density, use a) uniform distributions $x \sim U(-5, 5)$ and $u \sim U(0, M)$ with suitable M b) the density of the $N(0, 1)$ distribution.

3 Linear Models

Let us consider a linear model with p variables, $f(\mathbf{x}, \theta) = \theta_0 + \theta_1 \mathbf{x}_1 + \theta_2 \mathbf{x}_2 + \ldots + \theta_p \mathbf{x}_p$. Let us assume that we have noisy measurements $\mathbf{y} = (y_1, y_2, ..., y_n)$ obtained at points $\mathbf{x}_i = (x_{1i}, x_{2i}, ..., x_{ni})$ where $i = 1, ..., p$. Now, we can write the model in matrix notation:

$$\mathbf{y} = \mathbf{X}\theta + \varepsilon, \tag{26}$$

where $\mathbf{X}$ is the design matrix that contains the measured values for the control variables, augmented with a column of ones to account for the intercept term θ_0:

$$\mathbf{X} = \begin{pmatrix} 1 & x_{11} & x_{12} & \ldots & x_{1p} \\ 1 & x_{21} & x_{22} & \ldots & x_{2p} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & x_{n2} & \ldots & x_{np} \end{pmatrix}. \tag{27}$$

For linear models, we can derive a direct formula for the LSQ estimator. It can be shown (see exercises), that the LSQ estimate, that minimizes $SS(\theta) = ||\mathbf{y} - \mathbf{X}\theta||_2^2$, is obtained as the solution to the *normal equations* $\mathbf{X}^T\mathbf{X}\theta = \mathbf{X}^T\mathbf{y}$:

$$\hat{\theta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}. \tag{28}$$

In practice, it is often numerically easier to solve the system of normal equations than to explicitly compute the matrix inverse $(\mathbf{X}^T\mathbf{X})^{-1}$. In MATLAB, one can use the 'backslash' shortcut to obtain the LSQ estimate, `theta=X\y`, check out the code example given later in this section.

To obtain the statistics for the estimate, we can compute the covariance matrix $\mathrm{Cov}(\hat{\theta})$. Let us make the common assumption that we have independent and identically distributed measurement noise with measurement error variance σ^2. That is, we can write the covariance matrix for the measurement as $\mathrm{Cov}(\mathbf{y}) = \sigma^2\mathbf{I}$, where $\mathbf{I}$ is the identity matrix. Then, we can show (see exercises) that

$$\mathrm{Cov}(\hat{\theta}) = \sigma^2(\mathbf{X}^T\mathbf{X})^{-1}. \tag{29}$$

The diagonal elements of the covariance matrix give the variances of the estimated parameters, which are often reported by different statistical analysis tools.

Let us further assume that the measurement errors are Gaussian. Then, we can also conclude that the distribution of $\hat{\theta}$ is Gaussian, since $\hat{\theta}$ is simply a linear transformation of a Gaussian random variable $\mathbf{y}$, see equation (28). That is, the unknown parameter follows the normal distribution with mean and covariance matrix given by the above formulae: $\theta \sim \mathrm{N}(\hat{\theta}, \sigma^2(\mathbf{X}^T\mathbf{X})^{-1})$. That is, the probability density of the unknown parameter can be written as

$$p(\theta) = K \exp\left(-\frac{1}{2}(\theta - \hat{\theta})^T\mathbf{C}^{-1}(\theta - \hat{\theta})\right), \tag{30}$$

where $K = ((2\pi)^{d/2}|\Sigma|^{1/2})^{-1}$ is the normalization constant and $\mathbf{C} = \mathrm{Cov}(\hat{\theta})$.

3.1 Analyzing Modeling Results

In regression models, usually not all possible terms are needed to obtain a good fit between the model and the data. In empirical models, selecting the terms used is a problem of its own, and numerous methods have been developed for this model selection problem.

Selecting the relevant terms in the model is especially important for prediction and extrapolation purposes. It can be dangerous to employ a 'too fine' model. As a general principle, it is often wise to prefer simpler models and drop terms that do not significantly improve the model performance (A. Einstein: "a model should be as simple as possible, but not simpler than that"). Too complex models often result in 'over-fitting': the additional terms can end up explaining the random noise in the data. See exercises for an example of the effect of model complexity.

A natural criterion for selecting terms in a regression model is to drop terms that are poorly known (not well identified by the data). When we have fitted a linear model, we can compute the covariance matrix of the unknown parameters. Then, we can compute the 'signal-to-noise' ratios for the parameters in the model, also known as *t-values*. For parameter θ_i, the t-value is

$$t_i = \hat{\theta}_i / \mathrm{std}(\hat{\theta}_i), \tag{31}$$

where the standard deviations of the estimates can be read from the diagonal of the covariance matrix given in equation (29). The higher the t-value is, the smaller the relative uncertainty is in the estimate, and the more sure we are that the term is relevant in the model and should be included. We can select terms for which the t-values are clearly separated from zero. As a rule of thumb, we can require that, for instance, $|t_i| > 3$.

The t-values give an idea of how significant a parameter is in explaining the measurements. However, it does not tell much about how good the model fits the observations in general. A classical way to measure the goodness of the fit is the *coefficient of determination*, or R^2 value, which is written as

$$R^2 = 1 - \frac{\sum (y^i - f(x^i, \hat{b}))^2}{\sum (y^i - \bar{y})^2}. \tag{32}$$

The intuitive interpretation of the R^2 value is the following. Consider the simplest possible model $\mathbf{y} = \theta_0$, that is, the data are modeled by just a constant. In this case, the LSQ estimate is just the empirical mean of the data, $\hat{\theta}_0 = \overline{\mathbf{y}}$ (see exercises). That is, the R^2 value measures how much better the model fits to the data than a constant model. If the model is clearly better than the constant model, the R^2 value is close to 1.

The R^2 value tells how good the model fit is. However, it does not tell anything about the predictive power of the model, that is, how well the model is able to generalize to new situations. The predictive power of the model can be assessed, for instance, via cross-validation techniques. Cross-validation, in general, works as follows:

1. Leave out a part of the data from the design matrix $\mathbf{X}$ and response $\mathbf{y}$.

2. Fit the model using the remaining data.

3. Using the fitted model, predict the measurements that were left out.

4. Repeat steps 1-3 so that each response value in $\mathbf{y}$ is predicted.

The goodness of the predictions at each iteration can be assessed using the R^2 formula. The obtained number is called the Q^2 value and it describes the predictive power of the model better than R^2.

Cross-validation is commonly used for model selection. The idea is to test different models with different terms included and choose the model that gives the best Q^2 value. If the model is either too complex ('over-parameterized') or too crude, the model will predict poorly and give a low Q^2 value. In practice, the cross-validation procedure can be carried out by stepwise regression, which can be implemented in many ways, for instance by

- *Forward stepping*: test all terms individually, choose the one with highest Q^2 value. Continue by testing all remaining terms, together with those already chosen. On each step, choose the term with highest Q^2 value.
- *Backward stepping*: similarly, but starting with the full model, and dropping, one by one, the worst term on each iteration step.

Next, we give a short demonstration how linear regression can be computed with MATLAB and how the model fit and the significance of the parameters can be assessed.

Code example: linear regression, R^2- and t-values

Let us consider fitting the model $y = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_{11} x_1^2 + \theta_{12} x_1 x_2 + \theta_{22} x_2^2$ the to data given below:

$$\begin{bmatrix} x_1: & 100.0 & 220.0 & 100.0 & 220.0 & 75.1 & 244.8 & 160.0 & 160.0 & 160.0 & 75.1 & 75.1 \\ x_2: & 2.0 & 2.0 & 4.0 & 4.0 & 3.0 & 3.0 & 1.5 & 4.4 & 3.0 & 3.0 & 3.0 \\ y: & 25.0 & 14.0 & 6.9 & 5.9 & 14.1 & 9.3 & 18.2 & 5.6 & 9.6 & 14.9 & 14.8 \end{bmatrix}$$

The MATLAB code for fitting the model is given in `lin_fit.m`. First, we specify the data matrices and construct the design matrix by computing the second powers and interaction terms, and augmenting the matrix with a row of ones to account for the intercept in the model:

```
% Fitting a linear model to data
clear all; close all; clc;

%%%% The data:
```

```
X = [100.0  2.0
     220.0  2.0
     100.0  4.0
     220.0  4.0
     75.1   3.0
     244.8  3.0
     160.0  1.5
     160.0  4.4
     160.0  3.0
     75.1   3.0
     75.1   3.0];

Y = [25.0 14.0 6.9  5.9 14.1 9.3  18.2  5.6  9.6  14.9 14.8]';

n = length(Y);  % number of data points

% constructing the design matrix
X2 = [ones(n,1) X X(:,1).^2 X(:,1).*X(:,2) X(:,2).^2];
```

The LSQ fit is done by solving the normal equations (28), which in practice can be done with the backslash operator in MATLAB. Then, we estimate the measurement error variance using the repeated measurements (indices 5, 10 and 11 in the data), and compute the covariance matrix of the parameters using equation (29):

```
b     = X2\Y;   % LSQ fit

% estimating the covariance
repmeas = [5 10 11];    % indices of repeated measurements
sig = std(Y(repmeas));  % sigma estimate
cov_b  = sig^2*inv(X2'*X2);
```

Finally, we take the parameter standard deviations out from the covariance matrix and compute the t-values for the parameters, and the R^2-value for the fit. In addition, we graph the model fit:

```
% t-values
std_b  = sqrt(diag(cov_b));
t_b    = b./std_b

% R2 value
```

```
yfit  = X2*b;   % model response
R2 = 1-sum((Y-yfit).^2)/sum((Y-mean(Y)).^2)

% visualizing the fit
plot(1:n,Y,'o',1:n,yfit); title('model fit');
```

3.2 Design of Experiments

So far, we have assumed that the measurements for the design variables $\mathbf{X}$ and the response $\mathbf{y}$ are given. The task of Design of Experiments (DOE) is to figure out how to choose the experimental variables $\mathbf{X}$ so that maximal information about the unknown parameters θ is obtained with minimal experimental effort. While the DOE questions deserve a course of their own (at LUT there is one), we briefly present here some basic experimental design concepts for linear models.

In linear models, assuming i.i.d. measurement error with variance σ^2, the covariance matrix of the parameters is $\mathrm{Cov}(\hat{\theta}) = \sigma^2(\mathbf{X}^T\mathbf{X})^{-1}$. That is, the uncertainty in $\hat{\theta}$ depends on the noise level σ^2 and on the design matrix $\mathbf{X}$. The measurement noise level we cannot control, but the design points $\mathbf{X}$ where measurements are obtained can often be set. This leads to the question of design of experiments: how to choose $\mathbf{X}$ so that maximal information is obtained with minimal number of experiments. Note that in linear models the uncertainty of the parameters does not depend on the values of the unknown parameters that we wish to estimate (!). This suggests that, for linear models, one can derive general, 'case-independent' theory about how to choose $\mathbf{X}$. Here, we present the most common design plans for linear models, without considering the theory behind them.

Different design plans allow the fitting of different kinds of models. That is, the design of experiments should be selected according to the expected behavior of the model response:

- 2^N **design** enables the estimation of a model that contains first order terms and interaction terms. For instance, the two-parameter model $y = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_{12} x_1 x_2$ can be fitted using a 2^2 design.
- **Central composite design (CCD)** allows the estimation of a model that contains the quadratic terms. In a two-parameter example, we can fit $y = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_{11} x_1^2 + \theta_{12} x_1 x_2 + \theta_{22} x_2^2$.

Mathematically, the design plans are typically expressed in *coded units*, where the center point of the design is moved to the origin and the minimum and maximum are scaled to the values ± 1. If $\overline{x}_i$ denotes the mean value of factor i and Δ_i is the difference between the maximum and minimum values of the experimental region, the transformation from original units x_i to coded units X_i is given by

$$X_i = \frac{x_i - \overline{x}_i}{\Delta_i/2}. \tag{33}$$

The coded units give a generic way to present various design plans. In addition, scaling all variables to the same unit interval often results in more numerically stable computations.

The 2^N design puts measurements in the corners of the experimental region. In coded units, the design contains all combinations of levels ± 1. In addition to the corners, replicated measurements are usually performed at the center point of the experimental region to estimate the measurement noise.

The CCD design extends the 2^N plan by adding 'One Variable at a Time' (OVAT) measurements. That is, only the values of one variable are changed at a time while the others are fixed to the center point value. The OVAT measurements are placed so that they are in a circle centered at the center point with radius $\sqrt{2}$ in coded units. The CCD and 2^N designs are illustrated for $N = 2$ in Fig. 5.

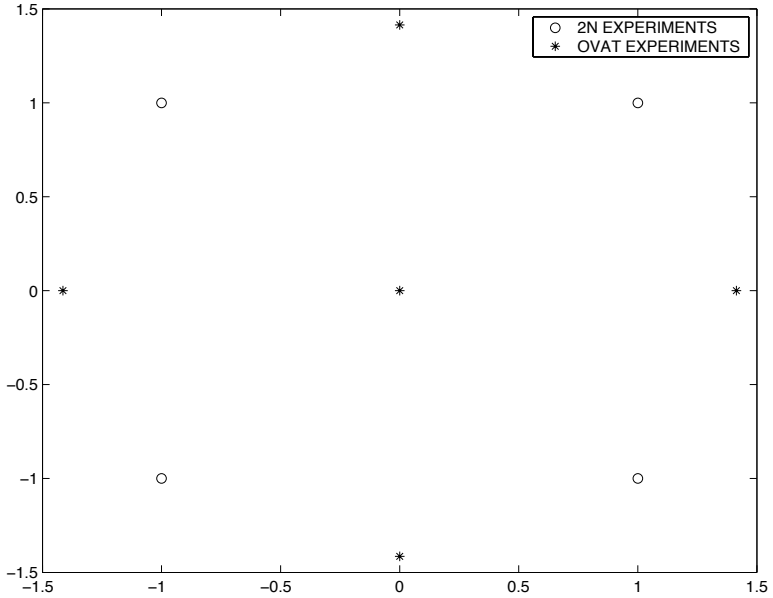

Figure 5: 2^N and CCD design plans in coded units.

Let us finally give an example for the case $N = 2$ (without any repeated measurements at the center point), which yields the following design matrices in coded units:

$$X_{2^N} = \begin{pmatrix} +1 & -1 \\ +1 & +1 \\ -1 & -1 \\ -1 & +1 \end{pmatrix} \qquad X_{CCD} = \begin{pmatrix} +1 & -1 \\ +1 & +1 \\ -1 & -1 \\ -1 & +1 \\ \sqrt{2} & 0 \\ -\sqrt{2} & 0 \\ 0 & \sqrt{2} \\ 0 & -\sqrt{2} \end{pmatrix} \tag{34}$$

Code example: coded units, 2^N and CCD designs

Let us here demonstrate how the most common design plans can be easily created with MATLAB using coded units, and how we can transform between coded units and real units. The demo is given in the program `doe_demo.m` and it uses three functions in the `utils` folder: the `code` function is used two make transforms between original and coded units, and `twon` and `composit` functions are used to generate the 2^N and CCD design plans (in coded units).

First, we initialize the program and generate two design matrices X_1 and X_2 that follow the 2^N and CCD design plans with 4 replicates at the center point. The `twon` and `composit` functions are called with two inputs: N (the number of variables) and the number of replicates:

```
% demonstrating coded units, 2N and CCD designs
clear all; close all;
addpath utils;

% two factors, 4 replications in the center, coded units
X=twon(2,4);        % 2N design
X2=composit(2,4);   % CCD design
```

Then, we demonstrate moving from coded units to real units, assuming that the experimental region of the two variables is $x_1 \in [100, 200]$ and $x_2 \in [50, 100]$. This is done with the `code` function, that has three inputs: the design matrix to be transformed, the minima and maxima for the design variables as a matrix (the experimental region) and the direction of the transform (-1 means transformation from coded units to real units, and $+1$ vice versa):

```
% moving from coded units to real units
% experimental area: 100<x1<200, 50<x2<100
minmax=[100 50;200 100];
X_real=code(X,minmax,-1);
X2_real=code(X2,minmax,-1);
```

Finally, we just print the designs in real units and in coded units, which results in the following output for the 2^N design:

```
2N design in coded and real units:

ans =

    -1    -1   100    50
     1    -1   200    50
    -1     1   100   100
     1     1   200   100
     0     0   150    75
     0     0   150    75
     0     0   150    75
     0     0   150    75
```

3.3 Exercises

1. This task was originally given in [Forsythe et al. 1977]. The data below describes how the population of the U.S. has developed from 1900 to 2000. Fit a kth order polynomial to the data with $k = 1, 2, ..., 8$ and predict with each fitted model how the population develops from 2000 to 2020. That is, first fit a 1st order polynomial $y = \theta_0 + \theta_1 x$, then a 2nd order polynomial $y = \theta_0 + \theta_1 x + \theta_2 x^2$ etc (here x is time).

$$\begin{pmatrix} \text{years after 1900} & \text{population (millions)} \\ 0 & 75.995 \\ 10 & 91.972 \\ 20 & 105.711 \\ 30 & 123.203 \\ 40 & 131.669 \\ 50 & 150.697 \\ 60 & 179.323 \\ 70 & 203.212 \\ 80 & 226.505 \\ 90 & 249.633 \\ 100 & 281.422 \end{pmatrix}$$

   Hint: use coded variables for fitting to avoid numerical difficulties, that is, scale the time variable to the interval $(-1, 1)$ (you can use the `code` function provided in the code package).

2. Show that the LSQ estimate for a constant model $\mathbf{y} = \theta_0$ is the mean of the data: $\hat{\theta}_0 = \overline{\mathbf{y}}$.

3. Fit a straight line $y = \theta_0 + \theta_1 x$ to the data given below. Compare the residuals to the standard deviation of the noise level, which you can estimate using the repeated measurements.

$$\begin{bmatrix} x: & 0.0 & 1.0 & 1.0 & 2.0 & 1.8 & 3.0 & 4.0 & 5.2 & 6.5 & 8.0 & 10.0 \\ y: & 5.00 & 5.04 & 5.12 & 5.28 & 5.48 & 5.72 & 6.00 & 6.32 & 6.68 & 7.08 & 7.52 \end{bmatrix}$$

4. Show that the formula (29) for the LSQ solution of a linear system holds.

5. Take the model and the data in the code example in Section 3.1. Try reducing the fitted model by dropping the least relevant terms in the model. Compare the R^2 values of the reduced model and the original model. You can start with the demo program `lin_fit.m`.

6. Consider the model $y = \theta_1 x_1 + \theta_2 x_2 + \varepsilon$, where the measurement noise $\varepsilon \sim \mathrm{N}(0, 1)$. Assume that the true parameter values are $\theta = (\theta_1, \theta_2) = (1, 2)$. Generate synthetic measurements by adding noise to the model solution with true parameter values, using the design points given below. Estimate θ with the simulated measurements. Repeat the data generation and estimation 1000 times (or more) and collect the obtained samples for the parameters. Compare the covariance matrices computed from the samples to the covariance matrix obtained from theory. Make 1D and 2D visualizations of the results.

$$\begin{bmatrix} x_1: & 1.0 & 1.0 & 2.0 & 1.8 & 3.0 & 4.0 & 5.2 & 6.5 & 8.0 & 10.0 \\ x_2: & 1.0 & 1.5 & 2.0 & 2.0 & 3.0 & 4.0 & 4.9 & 7.0 & 7.0 & 9.0 \end{bmatrix}$$

   The above design is an example of a bad design of experiments. Why? Make a better one: build a design matrix that gives more accurate results with the same number of experiments (or less).

7. Derive the LSQ estimate and its covariance for the correlated measurement error case $\mathbf{y} = \mathbf{X}\theta + \varepsilon$, where $\varepsilon \sim N(\mathbf{0}, \Sigma)$, by minimizing the weighted least squares expression $SS(\theta) = (\mathbf{y} - \mathbf{X}\theta)^T \Sigma^{-1} (\mathbf{y} - \mathbf{X}\theta)$. Hint: use the decomposition $\Sigma^{-1} = \mathbf{R}^T \mathbf{R}$ and transform the expression back to a non-weighted least squares problem $SS(\theta) = ||\tilde{\mathbf{y}} - \tilde{\mathbf{X}}\theta||_2^2$ and use the formulas (28) and (29).

4 Nonlinear Models

As seen in the previous section, direct formulas exist for computing LSQ estimates and their uncertainties for linear models. For nonlinear models, no such direct methods are available, and one has to resort to numerical methods and different approximations. The basic strategy, presented next, is to linearize the the nonlinear model and use the linear theory.

Let us consider a nonlinear model

$$\mathbf{y} = f(\mathbf{x}, \theta) + \varepsilon, \tag{35}$$

where $\mathbf{y}$ are the obtained measurements, $f(\mathbf{x}, \theta)$ is the model with design variables $\mathbf{x}$ and unknown parameters θ, and the measurement error is denoted by ε. To compute the LSQ estimate for the parameters, direct formulas are no longer available, and one has to numerically minimize the sum of squares

$$l(\theta) = \sum_{i=1}^{n} [y_i - f(x_i, \theta)]^2. \tag{36}$$

In practice, for most models one can use standard optimization routines implemented in computational software packages. For the purposes of this course, the MATLAB gradient-free nonlinear simplex optimizer `fminsearch` is enough, see the code examples in the end of this section.

Next, let us see how approximative error analysis can be performed for the parameters of a nonlinear model, based on linearizing the model at the LSQ estimate $\hat{\theta}$. The first three terms of the Taylor series expansion for the function $SS(\theta)$ at a point $\hat{\theta}$ can be written as

$$l(\theta) \approx l(\hat{\theta}) + \nabla l(\hat{\theta})^T(\theta - \hat{\theta}) + \frac{1}{2}(\theta - \hat{\theta})^T \mathbf{H} (\theta - \hat{\theta}), \tag{37}$$

where ∇ denotes the gradient and $\mathbf{H}$ is the Hessian matrix that contains the second derivatives. The second derivatives, or elements of the Hessian matrix, are

$$[\mathbf{H}]_{pq} = \frac{\partial^2 \ell(\hat{\theta})}{\partial \theta_p \partial \theta_q} = 2 \sum_{i=1}^{n} \frac{\partial f(x_i, \hat{\theta})}{\partial \theta_p} \frac{\partial f(x_i, \hat{\theta})}{\partial \theta_q} + 2 \sum_{i=1}^{n} (f(x_i, \hat{\theta}) - y_i) \frac{\partial^2 f(x_i, \hat{\theta})}{\partial \theta_p \partial \theta_q}. \tag{38}$$

Assuming that the residuals $f(x_i, \hat{\theta}) - y_i$ are small (they are minimized in the LSQ procedure), the Hessian matrix can be approximated using only first derivatives by dropping the residual terms:

$$[\mathbf{H}]_{pq} = \frac{\partial^2 \ell(\hat{\theta})}{\partial \theta_p \partial \theta_q} \approx 2 \sum_{i=1}^{n} \frac{\partial f(x_i, \hat{\theta})}{\partial \theta_p} \frac{\partial f(x_i, \hat{\theta})}{\partial \theta_q}. \tag{39}$$

The first derivatives can be collected into a matrix that is called the Jacobian matrix $\mathbf{J}$, which has elements

$$[\mathbf{J}]_{ip} = \frac{\partial f(x_i; \theta)}{\partial \theta_p}|_{\theta = \widehat{\theta}}, \tag{40}$$

where the notation means that the derivatives are evaluated at the estimate $\widehat{\theta}$. Now, the Hessian approximation can be written in a matrix form:

$$\mathbf{H} \approx 2\mathbf{J}^T\mathbf{J}. \tag{41}$$

Now, inserting this appoximation to the Taylor series expansion above, and noting that for the LSQ estimate the derivative of the sum of squares is zero, $\nabla l(\hat{\theta}) = \mathbf{0}$, we obtain

$$l(\theta) \approx l(\hat{\theta}) + (\theta - \hat{\theta})^T\mathbf{J}^T\mathbf{J}(\theta - \hat{\theta}). \tag{42}$$

Let us compare this to the linear case. For linear models, the least squares expression is

$$l(\theta) = ||\mathbf{y} - \mathbf{X}\theta||^2 = (\mathbf{y} - \mathbf{X}\theta)^T(\mathbf{y} - \mathbf{X}\theta) = \mathbf{y}^T\mathbf{y} - 2\mathbf{y}^T\mathbf{X}\theta + \theta^T\mathbf{X}^T\mathbf{X}\theta. \tag{43}$$

Differentiating the function twice gives the Hessian matrix $\mathbf{H} = \mathbf{X}^T\mathbf{X}$, and the Taylor series expansion can be written as

$$l(\theta) = l(\hat{\theta}) + (\theta - \hat{\theta})^T\mathbf{X}^T\mathbf{X}(\theta - \hat{\theta}). \tag{44}$$

Note that here we have an equality instead of an approximation, since higher order derivatives and the corresponding terms in the Taylor series expansion are zero.

Now we can compare the nonlinear and linear expressions in equations (42) and (44). We observe that in the linear approximation used in the nonlinear case, the Jacobian matrix $\mathbf{J}$ assumes the role of the design matrix $\mathbf{X}$ in the linear case. That is, the approximative error analysis for nonlinear models, assuming i.i.d. Gaussian errors with measurement error variance σ^2, is given by the covariance matrix

$$\mathrm{Cov}(\hat{\theta}) = \sigma^2(\mathbf{J}^T\mathbf{J})^{-1}. \tag{45}$$

The measurement error σ^2 can be estimated using repeated measurements. Often, however, replicated measurements are not available. In this case, the measurement noise can be estimated using the residuals of the fit, using the 'perfect model' assumption that *residuals* $\approx$ *measurement error*. An estimate for the measurement error can be obtained using the mean square error (MSE):

$$\sigma^2 \approx MSE = RSS/(n - p), \tag{46}$$

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. That is, the measurement error is computed as the average of the squared residuals, corrected by the number of estimated parameters. For more comprehensive theory behind approximation, see any classical statistics text book.

To sum up, we give a MATLAB code example that performs LSQ fitting and approximative error analysis based on the Jacobian matrix.

Code Example: Nonlinear LSQ Fitting and Approximative Error Analysis

Let us here consider a simple nonlinear model with two parameters, $\mathbf{y} = \theta_1(1 - \exp(-\theta_2\mathbf{x}))$. 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 $\mathbf{y} = (0.076, 0.258, 0.369, 0.492, 0.559)$.

We 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 do some initializations and call the `fminsearch` optimizer:

```
%%%%% LSQ fitting with the BOD model
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 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);
```

To estimate the uncertainty of the parameters, we use the formula $\mathrm{Cov}(\hat{\theta}) = \sigma^2(\mathbf{J}^T\mathbf{J})^{-1}$. The Jacobian here is computed analytically, which can be easily done for this simple model. Note, however, that usually computing the Jacobian analytically is difficult, and we need to approximate it numerically. An example of this follows later in this section.

The measurement error variance σ^2 is estimated using the MSE formula (46). Finally, we print the LSQ estimates, standard deviations and t-values, and plot the model fit.

```
%%%% 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 measurument noise estimated by the residuals
C = sigma2*inv(J'*J);
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))');
```

The parameter estimates found are $\hat{\theta} = (0.929, 0.104)$ and the t-values are 7.776 and 5.331. The figure produced by the code is given in Fig. 6.

Code Example: Parameter Estimation for Dynamical Models

In the example above, we estimated the parameters of a simple algebraic model. However, most mechanistic models used in practice are given as (a system of) differential equations for which no analytical solutions are often available. To solve the model equations and, for instance, compute the LSQ objective function, one has to use numerical solvers. In this section, we give an example of such a dynamical model, and demonstrate how model fitting works in practice in such a case.

As an example, let us consider again the chemical reactions $A \to B \to C$, which can be modeled as an ODE system as follows:

$$\frac{dA}{dt} = -k_1 A \tag{47}$$

$$\frac{dB}{dt} = k_1 A - k_2 B \tag{48}$$

$$\frac{dC}{dt} = k_2 B. \tag{49}$$

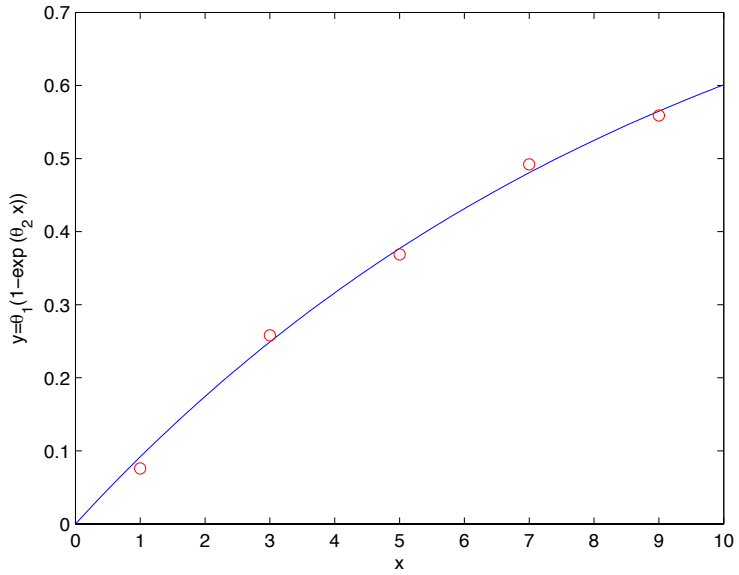

Figure 6: The model (blue line) fitted to the data (red circles).

The unknown parameters are the reaction rate coefficients, $\theta = (k_1, k_2)$. The data $\mathbf{y}$ consists of the values of the components A and B:

$$\begin{pmatrix} \text{time} & A & B \\ 1.0 & .504 & .415 \\ 3.0 & .217 & .594 \\ 5.0 & .101 & .493 \\ 7.0 & .064 & .394 \\ 9.0 & .008 & .309 \end{pmatrix}.$$

The initial values for the concentrations are $A(0) = 1$ and $B(0) = C(0) = 0$.

First, 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
```

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);
```

All MATLAB ODE solvers are used in the same way: the first input is the ODE function that specifies the ODE system that we are solving, followed by the solution time span and initial values. Then, the user can specify options that control how the

ODE solver works (again, empty matrix gives the default options). After options, the user can list extra variables that are possibly needed by the ODE function (here the parameter vector).

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 Fig. 7 reads as

---

```
% 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');
```

Code Example: Computing the Jacobian Numerically

In the BOD example given earlier in this section, the Jacobian was computed analytically to get the approximative statistics for the parameters. In real applications, this is seldom possible, since the forward model is often given as a numerical solution

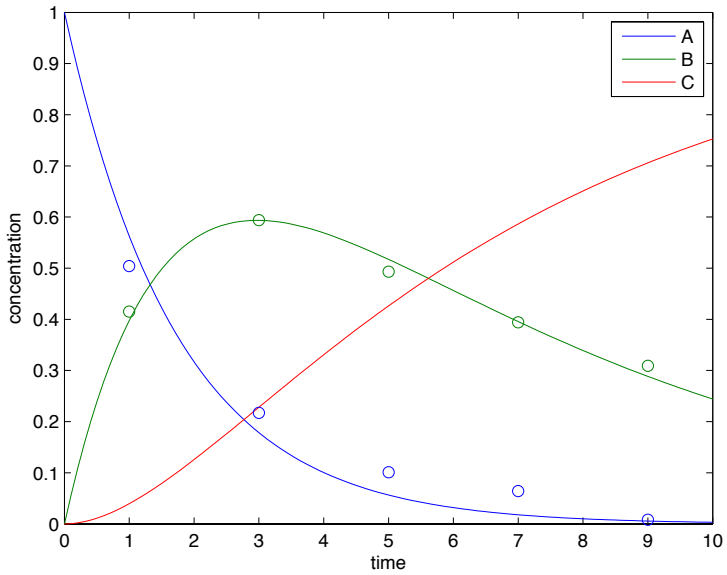

Figure 7: The model (lines) fitted to the data (circles).

to a differential equation, for instance. However, it is straightforward to compute numerical approximations for the derivatives in the Jacobian matrix, using, for instance, the two-sided finite difference formula

$$[\mathbf{J}]_{ip} = \frac{\partial f(x_i, \theta)}{\partial \theta_p}|_{\theta=\widehat{\theta}} \approx \frac{f(x_i; \theta + h) - f(x_i; \theta - h)}{2h}, \tag{50}$$

where h is a small difference added to the pth component of the θ vector. This finite difference approximation is implemented in the `jacob` function given in the `utils` folder, see `help jacob`. The function takes in the model function that evaluates $f(\mathbf{x}, \theta)$, the control variable values $\mathbf{x}$ and the point $\hat{\theta}$ where the Jacobian is computed.

An example of the Jacobian calculation is given in the program `jacob_demo.m`, using the chemical reaction model and data from the previous code example. Here, the Jacobian and the covariance matrix is computed as follows:

```
% numerical Jacobian
J=jacob(@ABCmodel,[0 data.time],thopt,[],data.y0);

% covariance estimate
sigma2=ssopt/(10-2);
C=sigma2*inv(J'*J);
```

The empty matrix in the `jacob` call means that we use the default values for the finite difference step sizes h (they can be also controlled, check the function help). Again, the zero point needs to be added to the time vector.

The results are visualized by drawing the 50% and 90% confidence ellipses for the parameters (the plotting code is not given here, check `jacob_demo.m`). The figure produced by the program is given in Fig. 8.

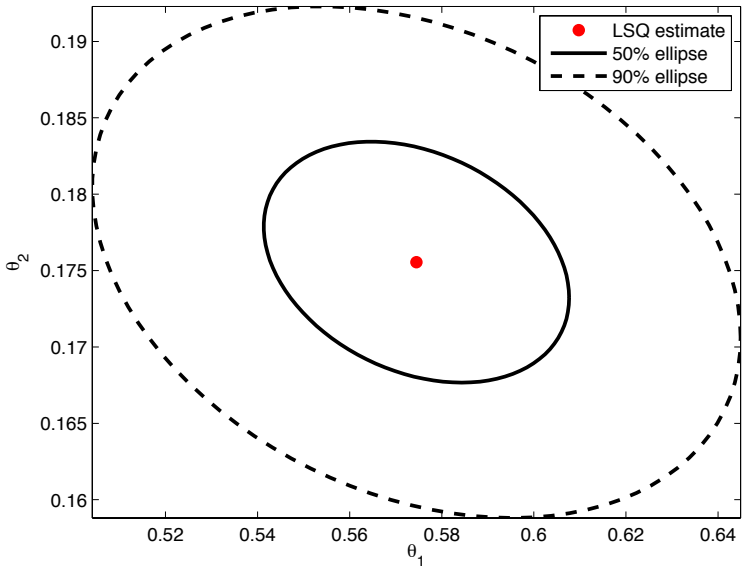

Figure 8: The LSQ estimates and two confidence ellipses.

4.1 Exercises

1. Fit the parameters k, p, y_0 of the logistic growth equation,

$$\frac{dy}{dt} = ky(p - y) \quad y(0) = y_0$$

to the data

$$\begin{pmatrix} \text{time} & y \\ 4.44 & 0.086 \\ 8.89 & 0.87 \\ 13.33 & 1.02 \\ 17.8 & 0.99 \\ 22.22 & 1.01 \end{pmatrix}$$

Compute the approximative covariance matrix of the parameters using the Jacobian of the model.

2. Beer cooling. At time $t = 0$, a glass of beer is at an initial temperature T_0. Beer will be cooled from outside by water, which has a fixed temperature $T_{\text{water}} = 5\ ^oC$. We measure the temperature of the beer at different times and get the following data:

$$\begin{bmatrix} 0 & 60 & 120 & 180 & 240 & 300 & 360 & 420 & 480 & 540 & 840 & 1020 & 1320 \\ 31 & 28 & 24 & 20 & 17.5 & 15.5 & 13.5 & 12 & 11 & 10 & 8 & 7 & 6.5 \end{bmatrix},$$

where the first row is time and the second row is temperature (in Celsius).

Note that the heat transfer takes place both through the glass and via the air/water surface, and a model for the beer temperature can be written as

$$dT/dt = -k_1(T - T_{\text{water}}) - k_2(T - T_{\text{air}}).$$

The temperature of the surrounding air is constant, $T_{\text{air}} = 23\ ^oC$. Estimate the unknown heat transfer coefficients $\theta = (k_1, k_2)$, and compute the approximative covariance matrix of the parameters.

3. Take the chemical reaction ODE system given in equations (47-49). Now, the reaction rates depend on the temperature T, and the dependency is modeled using the Arrhenius' equation

$$k_i(T) = k_{i,ref} \exp\left(-\frac{E_i}{R}\left(\frac{1}{T} - \frac{1}{T_{ref}}\right)\right), \quad i = (1, 2), \tag{51}$$

where $k_{i,ref}$ is the reaction rate at the reference temperature T_{ref}, E_i are the activation energies, T is the temperature and $R = 8.314$ is the gas constant. The goal is to estimate the parameters $\theta = (k_{1,ref}, E_1, k_{2,ref}, E_2)$ using the two batches of data given below, obtained at temperatures $T = 283K$ and $T = 313K$. The reference temperature you can basically choose, use for instance $T_{ref} = 300K$. Compute the LSQ estimate for the parameters.

Hints: reasonable starting values for the LSQ optimization are in the magnitude $k_{i,ref} \approx 1$ and $E_i \approx 10^4$. Note that now your LSQ function should simulate the model in two different temperatures, with reaction rates computed with the Arrhenius equation, and compare the simulation to the corresponding data.

$T = 282K$		
time	A	B
0	1.000	0.000
1	0.504	0.416
2	0.186	0.489
3	0.218	0.595
4	0.022	0.506
5	0.102	0.493
6	0.058	0.458
7	0.064	0.394
8	0.000	0.335
9	0.082	0.309

$T = 313K$		
time	A	B
0	1.000	0.000
1	0.415	0.518
2	0.156	0.613
3	0.196	0.644
4	0.055	0.444
5	0.011	0.435
6	0.000	0.323
7	0.032	0.390
8	0.000	0.149
9	0.079	0.222

5 Monte Carlo Methods for Parameter Estimation

The approximative error analysis for nonlinear models described in the previous section can sometimes given misleading results. An alternative way to obtain statistics for parameter estimates is to use various Monte Carlo (MC) random sampling methods. Before proceeding to Bayesian estimation and MCMC topics, which is the main focus of this course, we briefly present several 'classical' Monte Carlo methods that one can use to evaluate the uncertainty in $\hat{\theta}$.

5.1 Adding Noise to Data

Uncertainty in the model parameters θ in a model

$$\mathbf{y} = f(\mathbf{x}, \theta) + \epsilon \tag{52}$$

is caused by the noise ϵ. The LSQ fit with given data leads to a single estimated value $\hat{\theta}$. So, to obtain a distribution for values of θ, a natural idea might be to generate new data by adding random noise to the existing data and repeatedly fit different values $\hat{\theta}$ by the new data sets.

This approach is simple to implement and can work well, if the noise is correctly generated so that it agrees with the true measurement noise. However, often the structure of the noise is not properly known and generating new data is therefore questionable. Moreover, for a nonlinear model, a new iterative optimization needs to be performed after every time new data is generated, which can be time consuming. The results of the analysis are also dependent on the optimizer settings, like stopping tolerances.

5.2 Bootstrap

Bootstrapping is a very popular statistical analysis method, in spirit similar to the above basic MC method of generating new data and repeatedly computing the LSQ fit. However, in bootstrapping, no new data is generated, but new random combinations of the existing data are used. The basic idea of bootstrapping is given as a pseudo-algorithm below.

1. From the existing data $\mathbf{x} = (x_1, ..., x_n)$, $\mathbf{y} = (y_1, ..., y_n)$, sample new data $\tilde{\mathbf{x}}$, $\tilde{\mathbf{y}}$ with replacement. In practice, select n indices randomly from $1, ..., n$ (some may occur more than once) and choose the data points corresponding to the chosen indices.

2. Compute the fit using the resampled data $\tilde{\mathbf{x}}$, $\tilde{\mathbf{y}}$.

3. Go to step 1, until a desired number of θ samples are obtained.

Bootstrapping suffers from the same problems as the basic MC method of adding noise to data: it is dependent on the success of the optimization step and is computationally demanding since it requires repeated calls to an optimization routine.

Bootstrapping can also run into trouble, if the number of measurements is small: the resampled data might contain only a few points with which the parameter estimation problem is ill-posed (the data does not fix the parameters). To avoid this problem, bootstrapping is often implemented by resampling *residuals* instead of the original data points.

In *bootstrapping with residuals*, we first find the LSQ fit $\hat{\theta}$ for the model parameters, and then compute the residuals $r_i = y_i - f(x_i, \hat{\theta})$ for $i = 1, ..., n$. Then, in step 1 of the bootstrap algorithm, we generate a new set of residuals $(\tilde{r}_1, ..., \tilde{r}_n)$ by resampling the original residuals with replacement. Then, new data is generated by adding the sampled residuals to the fitted model responses: $\tilde{y}_i = f(x_i, \hat{\theta}) + \tilde{r}_i$, and this new data is used to compute the model fits in step 2 of the bootstrap algorithm.

5.3 Jack-knife

The jack-knife method is similar to the crossvalidation technique used for model selection (see Section 3.1), but without the prediction step. The technique can be used for estimating the variability of parameter estimates. Skipping the details, the jack-knife works as follows.

1 Leave out part of the data from the matrices $\mathbf{X}, \mathbf{Y}$

2 Fit the model parameters using the remaining data

Repeat the steps 1 and 2 so that each observation has been omitted. Compute estimates for the variability of the parameters.

5.4 Exercises

1. Generate data for the model $y = \theta_1(1 - \exp(-\theta_2 x)) + \epsilon$, supposing that the measurement noise is Gaussian (normally distributed) with $\epsilon \sim N(0, \sigma^2)$, $\sigma = 0.02$, using the parameter values obtained in the code example of Section 4 ($\theta_1 = 0.929$, $\theta_2 = 0.104$). Repeat the LSQ fitting some 1000 times (or more). Collect the parameter values and create the histograms for both parameters. Visualize also the 2-dimensional joint distribution of them, as well as the spread of the model predictions.

2. Find the distribution of the parameters θ_1, θ_2 of the model $y = \theta_1(1-\exp(-\theta_2 x))$ with data $x = (1, 3, 5, 7, 9)$, $y = 0.076, 0.258, 0.369, 0.492, 0.559$ by using a) bootstrap b) bootstrap for residuals. (Hint: indices for bootstrapping between 1 and n may be obtained by, e.g., the Matlab command `ceil(rand(n,1)*n)`. See also the function `bootstrp`).

3. Fit the model
$$y = \frac{\theta_1 x}{\theta_2 + x}$$
using the data $x = (28, 55, 83, 110, 138, 225, 375)$ and
$y = (0.053, 0.060, 0.112, 0.105, 0.099, 0.122, 0.125)$. Study the uncertainty of the solution by bootstrap.

6 Bayesian Estimation and MCMC

Let us consider the parameter estimation setting described in the previous chapters. In parameter estimation, parameters θ are estimated based on measurements $\mathbf{y}$, traditionally using, e.g., a least squares approach. In Bayesian parameter estimation, θ is interpreted as a random variable and the goal is to find the *posterior distribution* $\pi(\theta|\mathbf{y})$ of the parameters. The posterior distribution gives the probability density for values of θ, given measurements $\mathbf{y}$. Using the Bayes' formula, the posterior density can be written as

$$\pi(\theta|\mathbf{y}) = \frac{l(\mathbf{y}|\theta)p(\theta)}{\int l(\mathbf{y}|\theta)p(\theta)d\theta}. \tag{53}$$

The *likelihood* $l(\mathbf{y}|\theta)$ contains the measurement error model and gives the probability density of observing measurements $\mathbf{y}$ given that the parameter value is θ. For example, using the model $\mathbf{y} = f(\mathbf{x}, \theta) + \epsilon$ and employing a Gaussian i.i.d. error model, $\epsilon \sim N(0, \sigma^2\mathbf{I})$, and noting that $\epsilon = \mathbf{y} - f(\mathbf{x}, \theta)$, gives likelihood

$$l(\mathbf{y}|\theta) \propto \prod_{i=1}^{n} l(y_i|\theta) \propto \exp\left(-\frac{1}{2\sigma^2}\sum_{i=1}^{n}[y_i - f(x_i, \theta)]^2\right). \tag{54}$$

The *prior distribution* $p(\theta)$ contains all existing information about the parameters, such as simple bounds and other constraints. The integral $\int l(\mathbf{y}|\theta)p(\theta)d\theta$ in the denominator is the normalization constant that makes sure that the posterior $\pi(\theta|\mathbf{y})$ integrates to one.

Different point estimates can be derived from the posterior distribution. The Maximum a Posteriori (MAP) estimator maximizes $\pi(\theta|\mathbf{y})$ and the Maximum Likelihood (ML) estimator maximizes $l(\mathbf{y}|\theta)$. If the prior distribution is uniform within some bounds, ML and MAP coincide. With the Gaussian i.i.d. error assumption, ML coincides also with the classical Least Squares (LSQ) estimate, since minimizing the sum of squares term $SS(\theta) = \sum_{i=1}^{n}[y_i - f(x_i, \theta)]^2$ is equivalent to maximizing $l(\mathbf{y}|\theta)$ in the equation (54) above.

In principle, the posterior distribution gives the solution to the parameter estimation problem in a fully probabilistic sense. We can find the peak of the probability density, and determine, for instance, the 95% credibility regions for the parameters. However, working with the posterior density directly is challenging, since we need to compute the normalization constant in the Bayes formula, which is the integral $\int l(\mathbf{y}|\theta)p(\theta)d\theta$. In most cases this cannot be computed analytically, and classical numerical integration methods also become infeasible, if the number of parameters is larger than a few. With the so called Markov chain Monte Carlo (MCMC) methods, statistical inference for the model parameters can be done without explicitly computing this difficult integral.

MCMC methods aim at generating a sequence of random samples $(\theta_1, \theta_2, ..., \theta_N)$, whose distribution asymptotically approaches the posterior distribution as N increases. That is, the posterior density is not used directly, but samples from the posterior distribution are produced instead. The *Monte Carlo* term is used to describe methods that are based on random number generation. The sequence of

samples is generated so that each new point θ_{i+1} only depends on the previous point θ_i, and the samples therefore form a *Markov Chain*. Markov Chain theory can be used to show that the distribution of the resulting samples approach the correct target (posterior).

6.1 Metropolis Algorithm

One of the most widely used MCMC algorithms is the random walk Metropolis algorithm introduced already in 1950s in statistical physics literature [Metropolis et al. 1953]. The Metropolis algorithm is very simple: it works by generating candidate parameter values from a *proposal distribution* and then either accepting or rejecting the proposed value according to a simple rule. The Metropolis algorithm can be written as follows:

- 1 Initialize by choosing a starting point θ_1
- 2 Choose a new candidate $\hat{\theta}$ from a suitable **proposal distribution** $q(.|\theta_n)$, that may depend on the previous point of the chain.
- 3 **Accept** the candidate with probability

$$\alpha(\theta_n, \hat{\theta}) = \min\left(1, \frac{\pi(\hat{\theta})}{\pi(\theta_n)}\right). \tag{55}$$

  If rejected, repeat the previous point in the chain. Go back to step 2.

The Metropolis algorithm assumes a symmetric proposal distribution q, that is, the probability density of moving from the current point to the proposed point is the same as moving backwards from the proposed point to the current point: $q(\hat{\theta}|\theta_n) = q(\theta_n|\hat{\theta})$. A simple extension to non-symmetric proposal distributions is given by the *Metropolis-Hastings* algorithm, see Section 7.1. In this course we will use Gaussian proposals which are symmetric, and the Metropolis algorithm is therefore enough for our purposes.

One can see that in the Metropolis algorithm the candidate points that give a better posterior density value than the previous point (points where $\pi(\hat{\theta}) > \pi(\theta_n)$), or moves 'upward' in the posterior density function are always accepted. However, moves 'downward' may also be accepted, with probability given by the ratio of the posterior density values at the previous point and the proposed point. In code level, the accept-reject step can be implemented by generating a uniform random number $u \sim U(0,1)$ and accepting if $u \leq \pi(\hat{\theta})/\pi(\theta_i)$.

Note that in the Metropolis algorithm we only need to compute ratios of posterior densities. In the calculation of the ratio, the normalization constant (nasty integral) cancels out. This is what makes MCMC computationally feasible in multidimensional parameter estimation problems.

The problem remaining in the implementation of the Metropolis algorithm is defining the proposal distribution q. The proposal should be chosen so that it is easy to sample from and as 'close' to the underlying target distribution (posterior) as possible. An unsuitable proposal can lead to inefficient implementation:

- if the proposal is too large, the new candidates mostly miss the essential support π, they are chosen at points where $\pi \simeq 0$ and only rarely accepted.
- if the proposal is too small, the new candidates mostly are accepted, but from a small neighborhood of the previous point. So the chain moves only slowly, and may, in finite number of steps, not cover the target π.

In Fig. 9 below, MCMC chains for a single parameter with 3 different proposal sizes are given, illustrating how the effect of the proposal distribution is visible in the resulting samples.

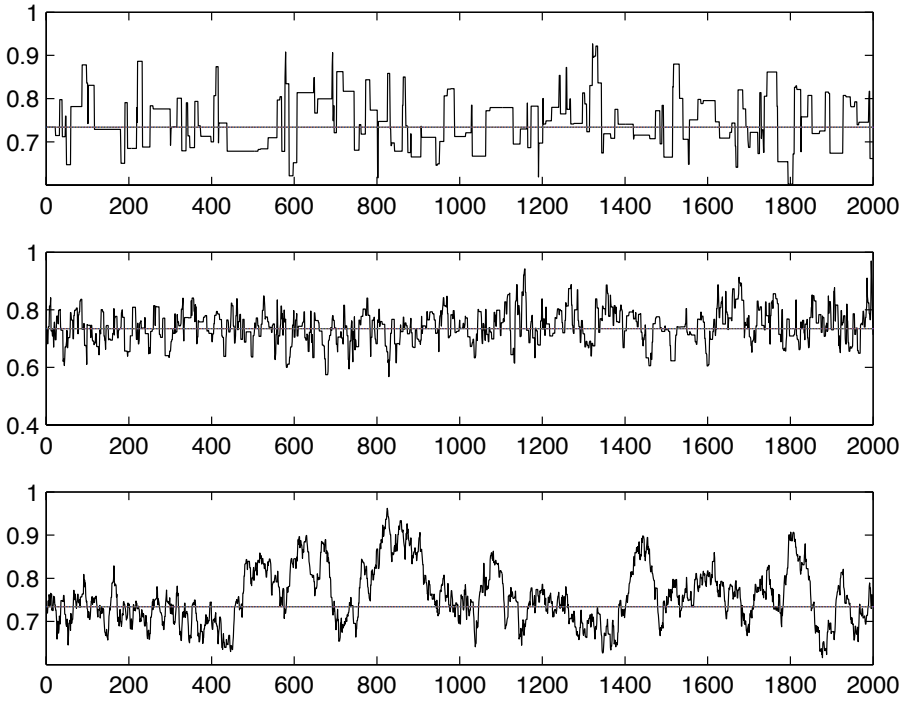

Figure 9: Examples of MCMC chains for one parameter. The upper picture tells that the proposal is too wide - the chain stays still for long periods. The lowest picture presents narrow proposal - the sampler explores the distribution slowly. The chain in the middle shows good 'mixing'.

In this course, we typically deal with multidimensional continuous parameter spaces. In such a setting, a multivariate Gaussian distribution is a common choice for a proposal distribution. In the commonly used *random walk Metropolis* algorithm, the current point in the chain is taken as the center point of the Gaussian proposal. The problem then is to find a suitable covariance matrix so that the size and the shape (orientation) of the proposal matches as well as possible with the target density (posterior) to produce efficient sampling. The covariance matrix selection issue is discussed in the next section.

The form of the posterior density depends on the case. Our most typical application is MCMC for standard nonlinear parameter estimation. Typically, the prior

information we have are some bound constraints for the parameters, and within the bounds we use a uniform, 'flat' prior, $p(\theta) \propto 1$. Assuming in addition independent measurement error with a known constant variance σ^2, the posterior density can be written as

$$\pi(\theta|\mathbf{y}) \propto l(\mathbf{y}|\theta)p(\theta) \propto \exp\left(-\frac{1}{2\sigma^2}SS(\theta)\right), \tag{56}$$

where $SS(\theta) = \sum_{i=1}^{n}[y_i - f(x_i, \theta)]^2$ is the sum of squares function that we minimize when the LSQ estimate is computed. Using this notation, the acceptance ration in the Metropolis algorithm reduces to

$$\alpha(\theta_n, \hat{\theta}) = \min\left(1, \frac{\pi(\hat{\theta})}{\pi(\theta_n)}\right) = \min\left(1, \exp\left(-\frac{1}{2\sigma^2}\left(SS(\hat{\theta}) - SS(\theta_n)\right)\right)\right). \tag{57}$$

Using these assumptions and this notation, the Metropolis algorithm with a multivariate Gaussian proposal distribution, with covariance matrix $\mathbf{C}$ and initial point $\theta_{old} = \theta_0$, can be written as follows:

1. Generate a candidate value $\theta_{new} \sim N(\theta_{old}, \mathbf{C})$ and compute $SS(\theta_{new})$.
2. Accept the candidate if $u < \exp\left(-\frac{1}{2\sigma^2}\left(SS(\theta_{new}) - SS(\theta_{old})\right)\right)$ where $u \sim U(0,1)$.
   - If accepted, add θ_{new} to the chain and set $\theta_{old} := \theta_{new}$ and $SS(\theta_{old}) := SS(\theta_{new})$.
   - If rejected, repeat θ_{old} in the chain.
3. Go to step 1 until a desired chain length is achieved.

This will be the version of the Metropolis algorithm that is mostly used in this course. For details about generating multivariate Gaussian random vectors, check Section 2.4. Note that although we assume here a flat prior, it is easy to add possible prior information about the parameters. In this course, we will mostly use bound constraints as prior information: for instance in chemical reaction modeling, reaction rate constants must be positive. Implementing simple bound constraints is easy: if the proposed parameter is out of bounds, it is simply rejected.

The progress of the Metropolis algorithm is animated in the `mcmcmovie` program for a two-dimensional non-gaussian 'banana-shaped' target distribution. A snapshot of the algorithm after 150 iterations is given in Fig. 10: the sampler starts from a poor initial point, eventually finds the target distribution and starts moving around the target.

Next, we will give a short MATLAB example how the Metropolis algorithm can be implemented.

Code example: implementing the Metropolis algorithm

Let us consider again the BOD example given in Section 4. That is, we fit the model $\mathbf{y} = \theta_1(1-\exp(-\theta_2\mathbf{x}))$ to data $\mathbf{x} = (1, 3, 5, 7, 9)$, $\mathbf{y} = (0.076, 0.258, 0.369, 0.492, 0.559)$.

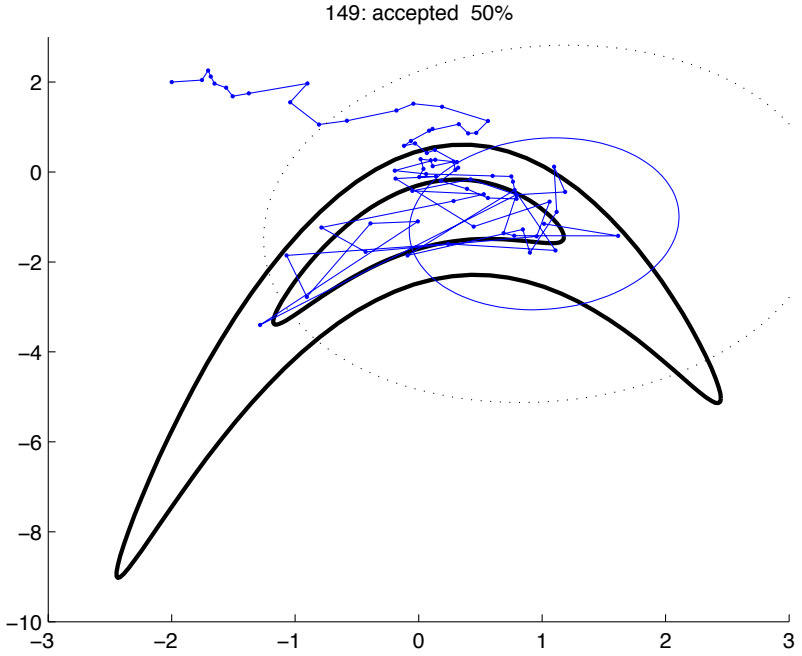

Figure 10: The path of the Metropolis sampler (blue line), when sampling a non-Gaussian target (contours given by black lines) with a Gaussian proposal distribution (ellipses).

We first compute the LSQ estimate by minimizing the sum of squares, and use the estimate as the starting point for the MCMC algorithm. A spherical proposal covariance $\mathbf{C} = \alpha\mathbf{I}$ is used, where α controls the size of the proposal distribution. The measurement error variance is estimated from the residuals using the MSE formula (46).

The code is given in two files, the main program `bod_mcmc.m` and the sum of squares objective function `bod_ss.m`. We start the main program as before by performing some initializations and minimizing the sum of squares:

```
%%%%% Metropolis MCMC algorithm
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 the LSQ estimate and estimate sigma**2 from the residuals
[bmin,ssmin]=fminsearch(@bod_ss,b_0,[],data);
```

```
sigma2 = ssmin/(n-2);
```

Then, we initialize the MCMC run, define (here just by hand) the covariance matrix of the proposal distribution and start the MCMC loop. Finally, we display the acceptance rate (fraction of accepted values) and graph the MCMC chains.

```
%%%% Generate the MCMC chain
nsimu  = 20000;
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;

%%%% start the simulation loop
for i=2:nsimu
   newpar = oldpar+randn(1,npar)*R;     % proposed new point
   ss2    = ss;                         % old SS
   ss1    = bod_ss(newpar,data);        % new SS
   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

% display the acceptance rate
```

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

%%%% 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;
```

The graphics produced by the program are given in Fig. 11. One 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. Selecting the proposal covariance is discussed in more detail in the next section.

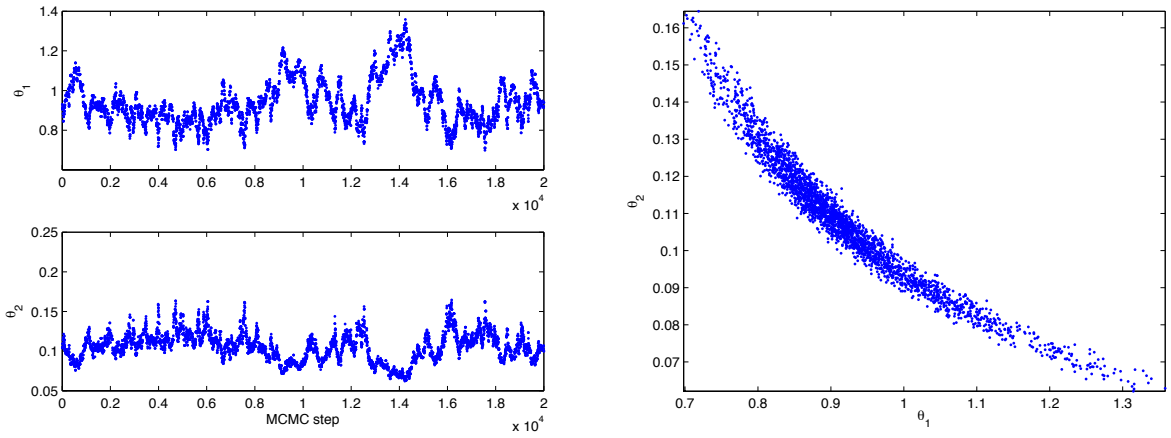

Figure 11: Left: the path of the MCMC sampler for both parameters. Right: the posterior distribution of the parameters.

One can see from this example that the basic Metropolis algorithm is very straightforward to implement in a programming environment: the MCMC loop itself only requires a few lines of code.

In addition to finding the distribution of the model parameters, the MCMC results can be used to simulate the distribution of any function of the parameters. For instance, one can study how the parameter uncertainty affects the model predictions simply by simulating the model with different possible parameter values given by MCMC. In this exampe, this *predictive distribution* can be computed and visualized with the following lines of code, which outputs Fig. 12:

```
% 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,'g-',x,y,'ro',xx,yfit,'k-');
xlabel('x'); ylabel('y=\theta_1(1-exp (\theta_2 x))');
title('MODEL PREDICTION DISTRIBUTION & DATA');
```

6.2 Selecting the Proposal Distribution

Selecting the proposal distribution is one of the main factors that affects the performance of an MCMC algorithm. In our typical setting, selecting the proposal means specifying the covariance matrix $\mathbf{C}$ of the multivariate Gaussian proposal distribution.

A good starting point for selecting $\mathbf{C}$ is to use the approximation of the covariance matrix obtained via linearization of the model, which was developed in Section 4. That is, we perform a Gaussian approximation of the posterior distribution at the LSQ estimate and choose the proposal covariance matrix $\mathbf{C} = \sigma^2(\mathbf{J}^T\mathbf{J})^{-1}$. This proposal can better match with the orientation of the target distribution, as illustrated for the previous example in Fig. 13.

In addition to orientation, scale of the proposal distribution is important. Theory has been developed for the optimal scaling of the proposal covariance matrix for the random walk Metropolis algorithms, see e.g. [Gelman et al. 1996]. It has been found that for Gaussian targets, an efficient scaling is $s_d = 2.4^2/d$, where d is the dimension of the problem (number of parameters). This result can be used as a

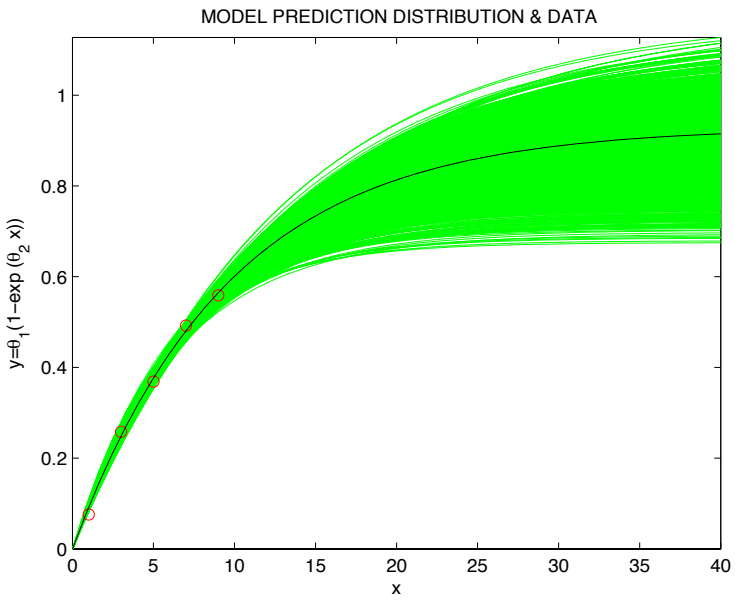

Figure 12: The predictive distribution of the model fit and prediction computed from the MCMC samples (green lines), the LSQ fit (black line) and the data points (red circles).

rule of thumb also for non-Gaussian targets. That is, utilizing the Jacobian-based covariance matrix, we can use proposal covariance matrix $\mathbf{C} = s_d\sigma^2(\mathbf{J}^T\mathbf{J})^{-1}$.

Next, we demonstrate how using the approximative covariance matrix affects the MCMC results in our simple example case.

Code example: Jacobian-based proposal covariance

Again, we take the model $\mathbf{y} = \theta_1(1 - \exp(-\theta_2\mathbf{x}))$ with data $\mathbf{x} = (1, 3, 5, 7, 9)$, $\mathbf{y} = (0.076, 0.258, 0.369, 0.492, 0.559)$. Now we take the proposal covariance matrix to be $\mathbf{C} = s_d\sigma^2(\mathbf{J}^T\mathbf{J})^{-1}$, where the Jacobian matrix is computed analytically. Note that usually it is cumbersome to compute the Jacobian analytically, and a numerical approximation for the Jacobian is used instead, as demonstrated in a code example in Section 4.

The main program is given in `bod_mcmc2.m` and the sum of squares function is the same as earlier, `bod_ss.m`. In the main program, we now compute the proposal covariance differently:

```
%%%% 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
```

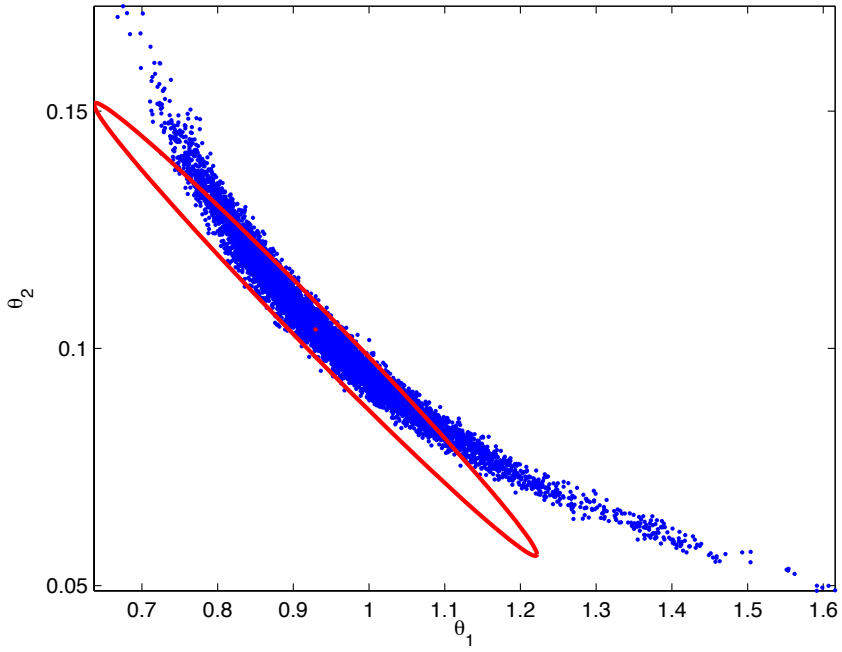

Figure 13: Posterior distribution obtained by MCMC (blue dots) and the 95% confidence ellipse corresponding to the proposal covariance $\mathbf{C} = \sigma^2(\mathbf{J}^T\mathbf{J})^{-1}$ (red line).

Otherwise, the MCMC code stays the same. The effect of the Jacobian-based proposal can be seen in the resulting plots, see Fig. 14. The mixing of the chain is improved a lot, compared to the spherical proposal in Fig. 11.

6.3 On MCMC Theory

The goal of this course is to be a practical introduction to statistical analysis methods, and therefore we will not go into the details of the theory behind MCMC methods. Here we simply give some central theoretical results that have been developed.

A central concept in MCMC theory is *ergodicity*, which guarantees the correctness of an MCMC algorithm in the sense that the Law of Large Numbers holds, and the averages computed from the MCMC samples approach the correct expected value as the number of samples increases. Formally, ergodicity is defined as follows. Let π be the density function of the target distribution in the d-dimensional Euclidean space $\mathbb{R}^d$. An MCMC algorithm is said to be ergodic if, for an arbitrary bounded and measurable function $f : \mathbb{R}^d \to \mathbb{R}$ and initial parameter value θ_0 that belongs to the support of π, it holds that

$$\lim_{n\to\infty} \frac{1}{n+1}(f(\theta_0) + f(\theta_1) + \ldots + f(\theta_n)) = \int_{R^d} f(\theta)\pi(\theta)d\theta, \tag{58}$$

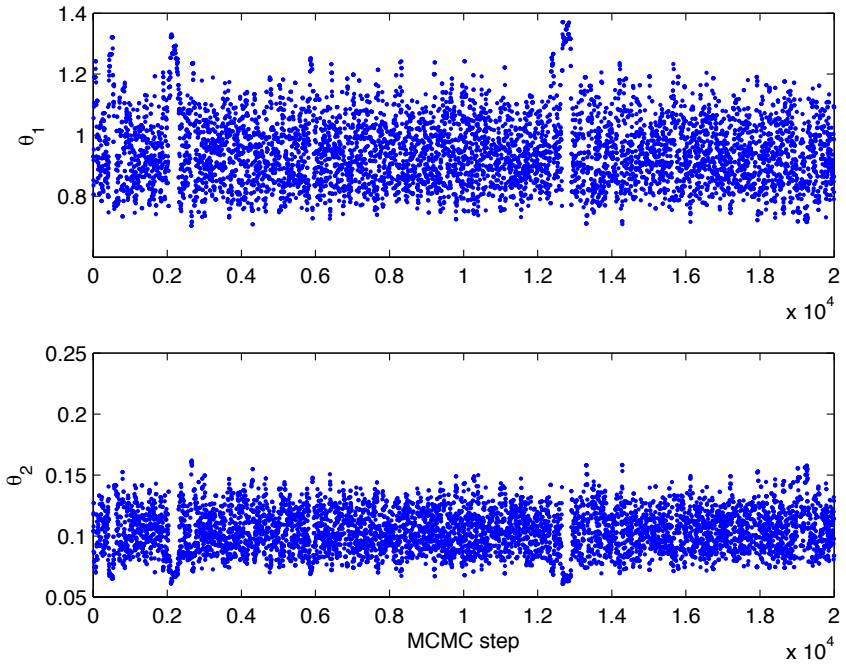

Figure 14: The path of the MCMC sampler for the two parameters with proposal covariance matrix $\mathbf{C} = s_d\sigma^2(\mathbf{J}^T\mathbf{J})^{-1}$.

where $(\theta_0, ..., \theta_n)$ are the samples produced by the MCMC algorithm. It can be shown, for instance, that the Metropolis algorithm described above is ergodic.

Intuitively, ergodicity (typically) means that, for a function that depends both on time and space, the time average at a fixed point in space equals the space average at a given time point. Here, the 'time' average – the left hand side of the above equation – is obtained via the discrete sampling by the algorithm, the 'space' average – the right hand side – by the integration over the probability distribution. The theorem simply states that the sampled values asymptotically approach the theoretically correct ones.

Note the role of the function f. If f is the characteristic function of a set A, i.e. $f(\theta) = 1$ if $\theta \in A$, $f(\theta) = 0$ otherwise, then the right hand side of the equation gives the probability measure of A, while the left hand side gives the frequency of 'hits' to A by the sampling.

But f might also be the model prediction $f(\mathbf{x}, \theta)$ made with parameter values θ. The theorem then states that the values calculated at the sampled parameters correctly gives the distribution of the model predictions, also known as the *predictive distribution*. An example of visualizing the predictive distribution was given in the code example earlier in this section, see Fig. 12.

More theoretical results can be derived for the convergence of MCMC. For instance, the ergodic theorem does not say about the rate of convergence, which is given by the Central Limit Theorem (CLT). For more details of MCMC convergence results, see MCMC text books, for instance [Brooks et al. 2011].

6.4 Adaptive MCMC

The bottleneck in MCMC computations is usually selecting a proposal distribution that matches well with the target distribution. The proposal covariance matrix using the linearization of the model discussed in the previous section is a good starting point, but does not always lead to efficient sampling. For instance, some of the parameters might be badly identified by the available data, which can result in a nearly singular Jacobian matrix and inefficient sampling. The purpose of adaptive MCMC methods is to tune the proposal 'on the run' as the sampling proceeds, using the information of the previously sampled points.

A simple way to implement adaptive MCMC is to simply compute the empirical covariance matrix of the points sampled so far, using equation (16), and use that as a proposal covariance matrix. Note that now the sampled points depend on the earlier history of the chain, not just the previous point, and the chain is therefore no longer Markovian. However, if the adaptation is based on an increasing part of the history, so that the number of previous points that is used to compute the empirical covariance matrix increases constantly as the sampling proceeds, it can be shown that the algorithm gives correct (ergodic) results, see [Haario et al. 2001].

6.4.1 Adaptive Metropolis

In the Adaptive Metropolis (AM) algorithm of [Haario et al. 2001], the proposal is taken to be Gaussian, centered at the current point, and the proposal covariance matrix is taken to be the empirical covariance matrix computed from the history. More precisely, if we have sampled points $(\theta_0, ..., \theta_{n-1})$, we propose the next candidate using the covariance $\mathbf{C}_n = s_d \mathrm{Cov}(\theta_0, \ldots, \theta_{n-1}) + \varepsilon \mathbf{I}_d$, where s_d is the scaling factor and $\varepsilon > 0$ is a regularization parameter that ensures that the proposal covariance matrix stays positive definite. In practice, ε can often be chosen to be very small or even set to zero.

In order to start the adaptation procedure, an arbitrary strictly positive definite initial covariance $\mathbf{C}_0$ is chosen, according to a priori knowledge (which may be quite poor). A time index $n_0 > 0$ defines the length of the initial non–adaptation period, which is often called the *burn-in* period in the literature, after which we use the empirical covariance matrix as the proposal. Thus, we let

$$\mathbf{C}_n = \begin{cases} \mathbf{C}_0, & n \leq n_0 \\ s_d \mathrm{Cov}(\theta_0, \ldots, \theta_{n-1}) + s_d \varepsilon \mathbf{I}_d, & n > n_0. \end{cases}$$

The initial proposal covariance $\mathbf{C}_0$ needs to be specified. A good starting point is often the approximative error analysis given by the linearization of the model. That is, we can use the scaled Jacobian-based covariance matrix as the initial proposal: $\mathbf{C}_0 = s_d \sigma^2 (\mathbf{J}^T \mathbf{J})^{-1}$. However, in most simple cases, it is enough to use a simple (e.g. diagonal) $\mathbf{C}_0$ that is small enough so that the sampler gets moving, and let the adaptation tune the proposal.

The empirical covariance matrix does not have to be recomputed every time, since recursive formulas exist. The empirical covariance matrix determined by

points $\theta_0, \ldots, \theta_k \in \mathbb{R}^d$ can be written in the form (check the details yourself)

$$\mathrm{Cov}(\theta_0, \ldots, \theta_k) = \frac{1}{k}\left(\sum_{i=0}^{k} \theta_i \theta_i^T \; - (k+1)\overline{\theta}_k \, \overline{\theta}_k^T\right),$$

where $\overline{\theta}_k = \frac{1}{k+1}\sum_{i=0}^{k} \theta_i$ is the empirical mean and the elements $\theta_i \in \mathbb{R}^d$ are considered as column vectors. The covariance matrix $\mathbf{C}_n$ satisfies the recursive formula

$$\mathbf{C}_{n+1} = \frac{n-1}{n}\mathbf{C}_n + \frac{s_d}{n}\left(n\overline{\theta}_{n-1}\overline{\theta}_{n-1}^T - (n+1)\overline{\theta}_n\overline{\theta}_n^T + \theta_n\theta_n^T + \varepsilon \mathbf{I}_d\right),$$

which permits the calculation of the covariance update with little computational cost (the mean, $\overline{\theta}_n$, also has an obvious recursive formula). Moreover, only the expression $\theta_n\theta_n^T/n$ is 'new' in the update formula, all the rest depends on previous mean values. So the effect of adaptation goes down as $1/n$; this is often called *diminishing adaptation*: in the long run, AM goes back to usual non-adaptive sampling, since new sampled points affect the proposal less and less as the sampling proceeds. This form of adaptation can proved to be ergodic. Note that the same adaptation, but with a *fixed* update length for the covariance, is *not ergodic*.

The choice for the length of the initial non–adaptive portion of the simulation, n_0, is free. The larger it is, the longer it takes for adaptation to start. It has been found that the adaptation might not be efficient if done at each time step, and one should adapt only at given time intervals. This form of adaptation improves the mixing properties of the algorithm, especially for high dimensions.

Finally, we can present the AM algorithm as the following pseudocode:

- Choose the lenght of the chain N and initialize θ_1 and $\mathbf{C}_1$ - for example, choose the θ_1 given by a LSQ fitting and take $\mathbf{C}_1$ as the approximative covariance calculated at θ_1 by linearization.
- For $k = 1, 2, ..., N$
  - Perform the Metropolis step, using proposal $N(\theta_k, \mathbf{C}_k)$.
  - Update $\mathbf{C}_{k+1} = \mathrm{Cov}(\theta_1, ..., \theta_k)$.

The algorithm may be implemented in several variations. One may compute the covariance by the whole chain $(\theta_1, ..., \theta_k)$ or by an increasing part of it, for instance $(\theta_{k/2}, ..., \theta_k)$. The covariance may also be updated only after a given number of steps k, instead of every step.

The AM algorithm performs well in a large variety of problems. In addition to AM, other more advanced adaptive MCMC methods have been developed. Next, we present one adaptive method that has turned out to be efficient in practice, the Delayed Rejection Adaptive Metropolis (DRAM) algorithm [Haario et al. 2006].

6.4.2 Delayed Rejection Adaptive Metropolis

In the regular Metropolis algorithm, a candidate move, $\tilde{\theta}_k$, is generated from a proposal distribution $q_1(\cdot|\theta_k)$ with the usual acceptance probability. In the *Delayed*

Rejection (DR) algorithm [Mira 2001], upon rejection, instead of repeating the previous value in the chain, $\theta_{k+1} = \theta_k$, a second stage move $\tilde{\theta}_k^{(2)}$ is proposed from a (possibly) different proposal distribution q_2. The second stage proposal is allowed to depend not only on the current position of the chain but also on what we have just proposed and rejected: $q_2(\cdot|\theta_k, \tilde{\theta}_k)$.

An ergodic chain is created, if the second stage proposal is accepted with suitably modified acceptance probability (details skipped here). The process of delaying rejection can be iterated to try sampling from further proposals. In practice, we often use only a 2-stage version, where the second stage proposal is a downscaled version of the first stage proposal. That is, upon rejection, we try a new candidate value closer to the current point.

The delayed rejection method can be combined with the adapting proposal covariance matrix. This algorithm, introduced in [Haario et al. 2006], is called Delayed Rejection Adaptive Metropolis (DRAM). The algorithm can be implemented in various ways, but we often use a the following simple implementation:

- The proposal at the first stage of DR is adapted just as in AM: the covariance matrix $\mathbf{C}_n^1$ for the Gaussian proposal is computed from the points of the sampled chain, no matter at which stage of DR these points have been accepted in the sample path.
- The covariance $\mathbf{C}_n^i$ of the proposal for the i-th stage ($i = 2, ..., m$) is always computed as a scaled version of the proposal of the first stage, $\mathbf{C}_n^i = \gamma_i \mathbf{C}_n^1$, with fixed scaling factors γ_i.

6.5 MCMC in practice: the `mcmcrun` tool

In this course we use a MATLAB code package that makes it easy to run MCMC analyses. The code package is written by Marko Laine, and it can be downloaded from `http://helios.fmi.fi/~lainema/mcmc/`. The toolbox provides a unified interface for specifying models, and implements, in addition to the basic Metropolis algorithm, the adaptive AM and DRAM methods described in the previous section. In this section, we will demonstrate the use of the toolbox using an example. This simple example demonstrates only the main features of the toolbox. For more documentation, see the web page and the appendix of the doctoral thesis of the author of the code [Laine 2008]. See also `help mcmcrun` for a short description of how the package is used.

Let us again consider the simple BOD model $\mathbf{y} = \theta_1(1 - \exp(-\theta_2 \mathbf{x}))$ with data $\mathbf{x} = (1, 3, 5, 7, 9)$, $\mathbf{y} = (0.076, 0.258, 0.369, 0.492, 0.559)$. The main program for running the MCMC analysis is `bod_mcmcrun.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 add the `mcmcstat` folder that contains the toolbox files to the MATLAB path:

```
%%%% MCMCRUN toolbox demo
```

```
clear all, close 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)`, see the file `bod_ss.m`. 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. Note that MATLAB can interpret `LaTex` and it is therefore easy to include mathematical expressions in the text.

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 = 20000;      % 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 given in Fig. 15, 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.

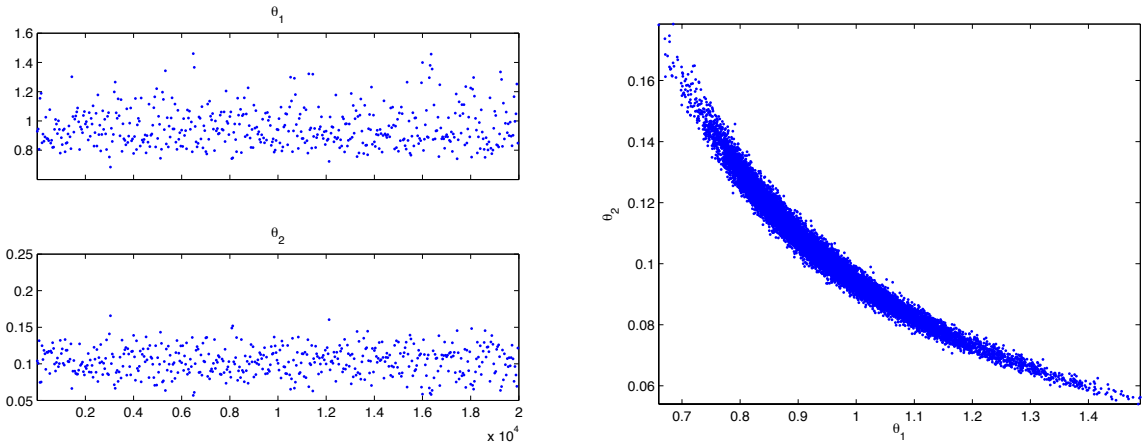

Figure 15: Left: the path of the MCMC sampler for both parameters. Right: the posterior distribution of the parameters.

6.6 Visualizing MCMC Output

The output of the MCMC algorithm can be visualized in many ways. Previously in this section we have plotted the output as two dimensional distribution plots and one dimensional 'chain' plots that give the sample paths for each parameter, see Fig. 11. In addition to these, one can obviously 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.

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 the BOD example used throughout this document, the pairs plots with density estimates with kernel width 1 can be obtained as follows:

```
mcmcplot(chain,[],results.names,'pairs',1);
```

In Fig. 16 below, the pairs plots are illustrated for two different values for the kernel width parameter. One can see that the kernel width affects the smoothness of the density estimates. Note that the function draws the 50% and the 95% confidence regions calculated based on the density estimation, and also estimates of the one dimensional marginal densities.

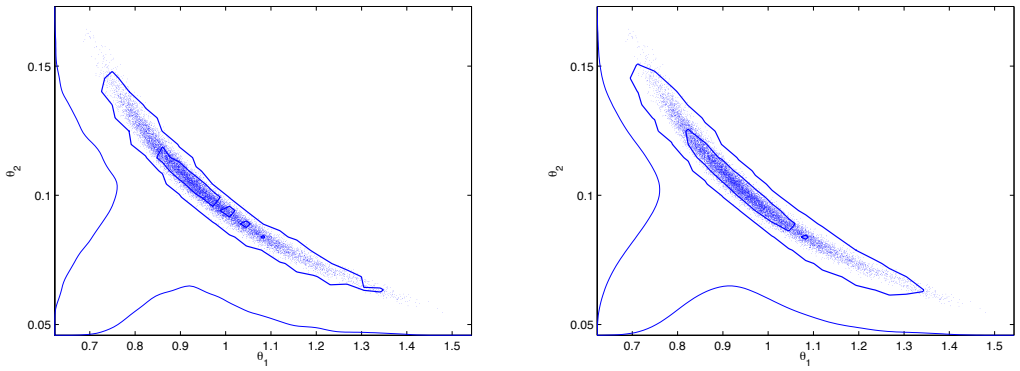

Figure 16: Pairs plots with two different width parameters for kernel density estimation.

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, as was demonstrated in Fig. 12.

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:

```
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.
```

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. The MCMC analysis with predictive distributions is given in the demo program `bod_mcmcrun_pred.m`. The program produces the figure given in Fig. 17.

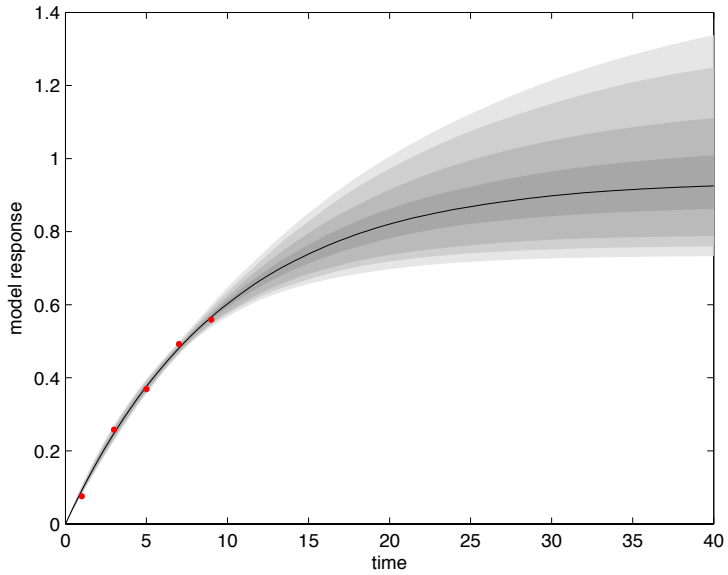

Figure 17: The data (red dots), the median of the predictions (black line) and the 50%, 95% and 99% confidence envelopes for the predictions (grey areas).

6.7 MCMC Convergence Diagnostics

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 BOD example, the function prints the following table:

```
>> chainstats(chain,results)
MCMC statistics, nsimu = 20000

                mean          std       MC_err          tau       geweke
--------------------------------------------------------------------------
  \theta_1     0.9615      0.13414   0.0048443       35.481      0.99295
  \theta_2     0.1027     0.018925  0.00055183       26.766      0.98571
--------------------------------------------------------------------------
```

That is, we get the mean and the standard deviation of the chain. In addition we get an estimate of the *Monte Carlo standard error* of the mean of the parameters, the integrated autocorrelation time and the Geweke convergence diagnostic, respectively. Check MCMC textbooks for details of these statistics.

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 $k = 1, ..., 100$ can be plotted as follows:

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

The code produces the autocorrelation plots for both parameters, given in Fig. 18. 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.

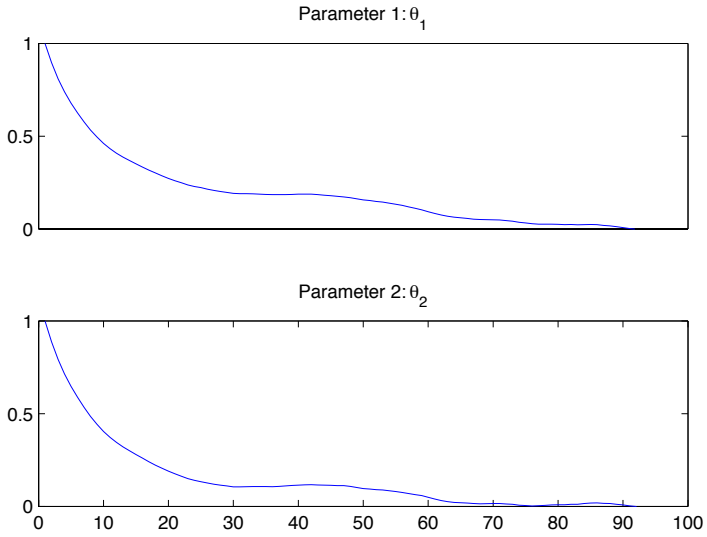

Figure 18: The autocorrelation function for two parameters.

6.8 Exercises

In the exercises below, you can use your own MCMC implementation, or use the `mcmcrun` tool demonstrated in the previous section.

1. Use the Metropolis MCMC algorithm to find the distribution of the parameters (θ_1, θ_2) of the model

$$\mathbf{y} = \frac{\theta_1 \mathbf{x}}{\theta_2 + \mathbf{x}} \tag{59}$$

with $\mathbf{y} = (0.053, 0.060, 0.112, 0.105, 0.099, 0.122)$, $\mathbf{x} = (28, 55, 110, 138, 225, 375)$. Study the impact of different proposal distributions (non-correlated Gaussian with different variances, a Gaussian with covariance computed by a previously sampled chain, covariance as calculated by the Jacobian of the LSQ fit). Visualize the parameter distributions and the predictive distribution.

2. Take the model and data from exercise 1 in Section 4. Estimate the parameters using MCMC and compare the results to the classical, Jacobian-based analysis.

3. Take the model and data from exercise 2 in Section 4. Estimate the parameters using MCMC. Visualize the uncertainty in the parameters.

4. Take the model and data from exercise 3 in Section 4. Estimate the parameters using MCMC. Visualize the uncertainty in the parameters and in the model response.

5. To study a chemical reaction $A+B \longrightarrow C$ the concentration of the component A was analyzed in a case where the reaction was started with the initial values $A = 0.054, \quad B = 0.106$ at $t = 0$. The data below was obtained

$$\begin{pmatrix} \text{time} & A \\ 426 & 0.0351 \\ 1150 & 0.0222 \\ 1660 & 0.0186 \\ 3120 & 0.0124 \end{pmatrix}$$

The reaction rate was written in the form $k = \theta A^{n_1} B^{n_2}$. Write an ODE system for the kinetics. Fit the unknown parameters θ, n_1, n_2 to the data. Study, by MCMC runs, to which extent the parameters are identifiable by this data. Hint: start by fixing one or both of the 2 parameters n_1, n_2 to integer values 1 or 2. Be prepared to have some numerical difficulties!

7 Further Monte Carlo Topics

7.1 Metropolis-Hastings Algorithm

In the Metropolis algorithm presented in Section 6.1, the proposal distribution was assumed to be symmetric, that is, for two parameter values θ_1 and θ_2 we have $q(\theta_2|\theta_1) = q(\theta_1|\theta_2)$, where $q(\theta_2|\theta_1)$ denotes the density for proposing a move from θ_1 to θ_2. However, the algorithm can be extended for non-symmetric proposal distributions, which was developed in [Hastings 1970].

In practice, the algorithm is otherwise the same as the Metropolis algorithm, but the acceptance probability is slightly modified to account for the non-symmetry. In the Metropolis-Hastings algorithm (MH), the probability of accepting the move from θ_n to $\hat{\theta}$ is given by

$$\alpha(\theta_n, \hat{\theta}) = \min\left(1, \frac{\pi(\hat{\theta})q(\theta_n|\hat{\theta})}{\pi(\theta_n)q(\hat{\theta}|\theta_n)}\right). \tag{60}$$

Comparing to the acceptance probability of the Metropolis algorithm in equation (55), one can see that the only difference is the inclusion of the ratio of the proposal densities, $q(\theta_n|\hat{\theta})/q(\hat{\theta}|\theta_n)$.

In this course, the MH algorithm has little practical relevance, since we use symmetric Gaussian proposal in the example cases.

7.2 Gibbs Sampling

In the Metropolis algorithm presented above, candidate values for all parameters are proposed at the same time. However, sometimes, especially in high-dimensional problems, it may be difficult to find a good multivariate proposal distribution for all parameters simultaneously. The idea in Gibbs sampling is to reduce the sampling to one dimensional distributions: each parameter is sampled in turn, while the other parameters are kept fixed.

In more detail, the parameter vector $(\theta_1, \theta_2, ..., \theta_p)$ is updated in sweeps, by updating one coordinate at a time. This may be done if the 1–dimensional *conditional distributions* $\pi(\theta_i|\theta_1, ..., \theta_{i-1}, \theta_{i+1}, ..., \theta_p)$ are known. In many cases these reduce to simple known densities which are easy to sample from. Often, however – in non-linear problems – the conditional distributions are not known, and they must be approximated by computing 'sufficiently' many values in the 1D directions.

Gibbs sampling as a pseudocode that creates a chain of length N can be written as follows:

- for k=1,...,N
  - for i=1,2,...,p
    sample θ_i^k from the 1D conditional distribution $\pi(\theta_i|\theta_1, ..., \theta_{i-1}, \theta_{i+1}, ..., \theta_p)$.

Note that there is no accept-reject procedure here and the point taken from the 1D distribution is always accepted, but the creation of the 1D (approximative) distribution may require several evaluations of the objective function, if the conditional distributions do not assume any known simple form.

If the 1D distribution for θ_k is not known, it must be approximatively created. This may be done by evaluating $\pi(\theta_i|\theta_1, ..., \theta_{i-1}, \theta_{i+1}, ..., \theta_p)$ with respect to the coordinate i a given number of times. The computed values can then be used to create an empirical CDF. The new value for θ_i^k can then be sampled from the empirical distribution by using the inverse CDF method: sample a random point uniformly on $[0, 1]$ and compute the inverse of the above empirical CDF at that point.

7.3 Component-wise Metropolis

Instead of sampling directly from the one-dimensional conditional distributions, as in Gibbs sampling, one can perform component-wise Metropolis sampling. The proposal distribution of each component is, for instance, a normal distribution with the present point as the center point and with a given variance, separate for each coordinate. The coordinates are updated in loops, similarly as in Gibbs sampling.

Let again $\pi(\theta)$ denote the density of our target distribution in a d dimensional Euclidean space, typically a posterior density distribution which we can evaluate up to a normalizing constant. The sequence $(\theta_0, \theta_1, \ldots)$ denotes the full states of the Markov process, that is, we consider a new state updated as soon as all the d coordinates (or components) of the state have been separately updated. When sampling the i:th coordinate θ_t^i $(i = 1, \ldots, d)$ of the t:th state θ_t we apply the standard 1-dimensional Metropolis step:

1. Sample z^i from 1-dimensional normally distributed proposal distribution $q_t^i \sim N(\theta_{t-1}^i, v^i)$ centered at previous point with variance v^i.

2. Accept the candidate point z^i with probability

$$\min\left(1, \frac{\pi(\theta_t^1, \ldots, \theta_t^{i-1}, z^i, \theta_{t-1}^{i+1}, \ldots, \theta_{t-1}^d)}{\pi(\theta_t^1, \ldots, \theta_t^{i-1}, \theta_{t-1}^i, \theta_{t-1}^{i+1}, \ldots, \theta_{t-1}^d)}\right),$$

   in which case set $\theta_t^i = z^i$, and otherwise $\theta_t^i = \theta_{t-1}^i$.

Note that, after a full loop over the coordinates, obtaining a changed value for θ_t typically is more likely than in standard Metropolis - since each coordinate may separately be accepted with a reasonable high probability. On the other hand, the CPU time needed for each coordinate loop increases with the dimension d.

The idea of proposal adaptation can be also extended to the single component sampling. This gives the *Single Component Adaptive Metropolis* (SCAM) method developed in [Haario et al. 2005], which can be useful in higher dimensional problems.

7.4 Metropolis-within-Gibbs

The component-wise Metropolis and Gibbs sampling algorithms can be combined: some coordinates may be updated by Gibbs sampling and some with Metropolis accept-reject steps. This approach is sometimes called Metropolis-within-Gibbs sampling. For coordinates that assume simple forms so that the direct sampling is

easy, Gibbs sampling may be preferable, and in other cases one can apply Metropolis steps.

7.5 Importance Sampling

Importance sampling is a Monte Carlo method for approximating integrals of form

$$\mathrm{E}_p(f) = \int f(x)p(x)dx, \tag{61}$$

that is, computing expectations of a function $f(x)$ with respect to a distribution given by the density function $p(x)$. A straightforward way would be to sample points $x_1, ..., x_m$ from the distribution given by $p(x)$ and approximate then the expectation by the average

$$\mathrm{E}_p(f) \approx \bar{f}_m = \frac{1}{m}\sum_{j=1}^{m} f(x_j). \tag{62}$$

Suppose that we can not (or it is 'expensive') sample directly from $p(x)$, but we do know a density function $g(x)$ such that $g(x) > 0$ if $p(x) > 0$, and sampling from the density $g(x)$ is easier. By the identity

$$E_f(p) = \int f(x)\frac{p(x)}{g(x)}g(x)dx \tag{63}$$

(where we take the integrand as zero if both p and g vanish) we may sample from $g(x)$ and approximate the expected value as

$$\bar{f}_m = \frac{1}{m}\sum_{j=1}^{m} f(x_j)\frac{p(x_j)}{g(x_j)} = \frac{1}{m}\sum_{j=1}^{m} f(x_j)w(x_j). \tag{64}$$

The function g is referred to as the *importance function* and w as the *importance weight*. The function g should be chosen so that it mimics the distribution p, and is easy to sample from. Naturally, the main problem here is how to find a proper importance function for each problem.

7.6 Conjugate Priors and MCMC Estimation of σ^2

In our typical applications, the likelihood distribution reads as

$$l(\mathbf{y}|\theta) = (2\pi\sigma^2)^{-n/2}\exp\left(-\frac{1}{2\sigma^2}SS(\theta)\right), \tag{65}$$

where $SS(\theta) = \sum_{i=1}^{n}[y_i - f(x_i, \theta)]^2$ is the sum of squares function. In practice, we need to specify the value for the measurement error variance σ^2. Previously, we have given a fixed value for σ^2, estimated from the residuals of the model fit or from repeated measurements. However, instead of fixing σ^2 to a specific point estimate, we can also regard σ^2 as a random variable and treat it in a Bayesian way by sampling it along with the model parameters in the MCMC algorithm.

We often have a rather good idea about the level of the measurement error, and we therefore would like to specify a prior distribution for it. A computationally convenient choice for the prior is obtained using the *conjugacy* property. If we set the prior so that the posterior is of the same form as the prior, the prior is called a *conjugate prior*.

Looking at the Gaussian likelihood (65), and considering it as a function of σ^2 (with fixed θ), we see that the inverse variance $1/\sigma^2$ has a Gamma type distribution (check, e.g., Wikipedia for the density function of the Gamma distribution). The conjugate prior for the Gamma distribution is also Gamma. That is, if we specify a Gamma prior for $1/\sigma^2$, the conditional posterior $p(\sigma^{-2}|\mathbf{y}, \theta)$ will also be Gamma distributed. More specifically (check, e.g., [Gelman et al. 1996] for details), the prior for σ^{-2} can be defined as

$$\sigma^{-2} \sim \Gamma(\frac{n_0}{2}, \frac{n_0}{2} S_0^2). \tag{66}$$

That is, we define the prior for the measurement error variance with two parameters, n_0 and S_0^2. This parameterization is chosen because the prior parameters are easy to interpret: S_0^2 gives the mean value for σ^2 and n_0 defines how accurate we think the value S_0^2 is. The higher n_0 is, the more peaked the prior distribution is around S_0^2, and the more informative the prior is. In Fig. 19 below, the prior density for σ^2 is illustrated.

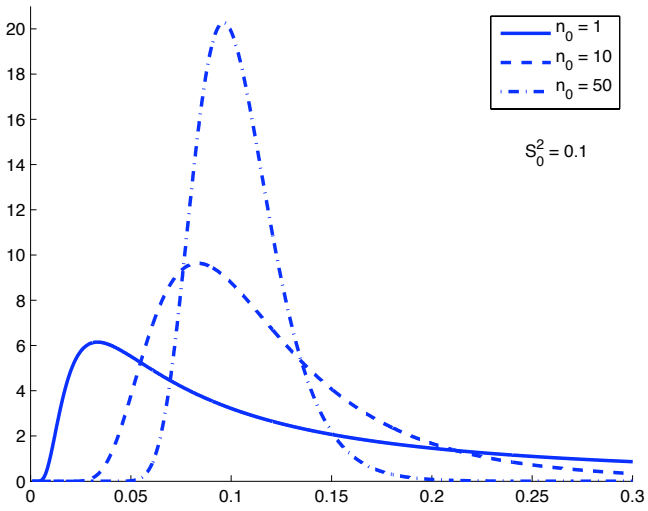

Figure 19: Prior densities for error variance with $S_0^2 = 0.1$ and different values for n_0.

With the conjugate prior, we can derive a Gamma posterior for the conditional posterior for the error variance, $p(\sigma^{-2}|\mathbf{y}, \theta)$. Without going into the details (try to work them out yourself), the conditional posterior for σ^{-2} posterior can be written as

$$p(\sigma^{-2}|\mathbf{y}, \theta) = \Gamma(\frac{n_0 + n}{2}, \frac{n_0 S_0^2 + SS(\theta)}{2}). \tag{67}$$

Now we have an analytical expression for the conditional distribution of σ^{-2}, and we can build a Gibbs sampler that first samples θ as usual and then samples a new value for σ^2 from the above density by iterating the following steps:

1. Sample a new θ value from $p(\theta|\sigma^{-2}, \mathbf{y})$
2. Sample a new σ^2 value from $p(\sigma^{-2}|\mathbf{y}, \theta)$

The `mcmcrun` tool used in this course contains the σ^2 sampling feature presented above. Next, we give a code example of how this works.

Code example: σ^2 sampling via `mcmcrun`

This demo uses again the BOD model $\mathbf{y} = \theta_1(1 - \exp(-\theta_2\mathbf{x}))$ with data $\mathbf{x} = (1, 3, 5, 7, 9)$, $\mathbf{y} = (0.076, 0.258, 0.369, 0.492, 0.559)$. The program is given in `bod_mcmcrun_sig.m`, which is only a slight modification to `bod_mcmcrun.m` presented earlier (where σ^2 was fixed). In the `mcmcrun` code, the σ^2 sampling is turned on using the `options` structure by setting

```
options.updatesigma = 1;    % sample sigma2 too
```

The conjugate prior is controlled with two parameters, n_0 and S_0^2, which can be set in the `model` structure. If they are not set, the defaults will be used ($n_0 = 1$ and the `model.sigma2` value for S_0^2). Here, we use the previously calculated MSE estimate as S_0^2 and $n_0 = 5$:

```
model.S20=sigma2;       % prior for sigma2
model.N0=5;             % prior accuracy for sigma
```

Other than these lines, the MCMC part of the code remains the same. However, in the call to `mcmcrun`, we now take an extra output variable, the `s2chain` that contains the sampled σ^2 values. In addition, we can also take the chain of sum of squares values out from the function:

```
% calling MCMCRUN
[results,chain,s2chain,sschain] = mcmcrun(model,data,params,options);
```

In the visualization part of the code, which gives Fig. 20, we plot the sampled σ^2 chain and compute the histogram of the samples:

```
% visualizing the results using MCMCPLOT
figure(1);
mcmcplot(s2chain);
title('\sigma^2 chain');

figure(2);
mcmcplot(s2chain,[],[],'hist');
title('\sigma^2 histogram');
```

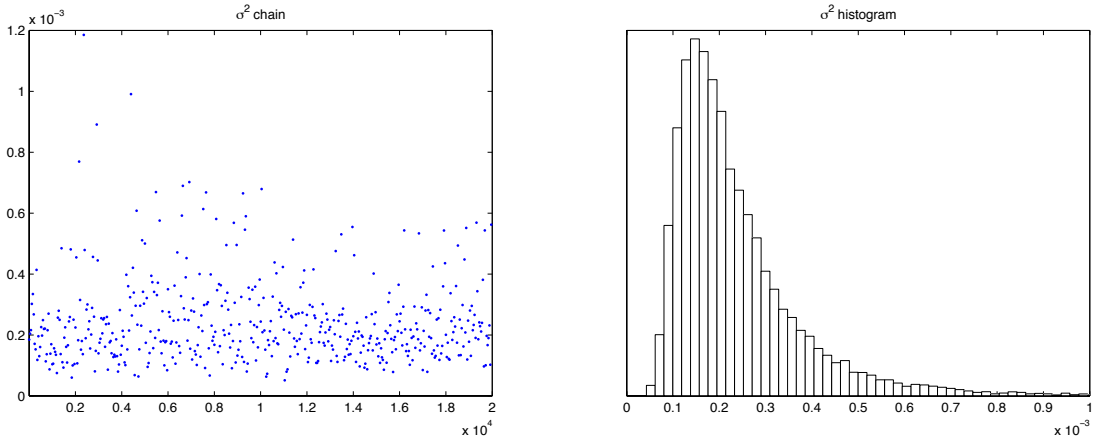

Figure 20: Left: the MCMC chain for σ^2. Right: the histogram from σ^2 samples.

7.7 Hierarchical Modeling

This section is under construction.

7.8 Exercises

1. Consider the task of calculating the tail probability $P(X > M)$, where $X \sim N(0, 1)$ and $M = 4.5$. Compare two Monte Carlo approaches:

   (a) the basic MC approach, where you sample numbers from $N(0, 1)$ and calculate the fraction of points satisfying $X > M$.

   (b) the importance sampling approach with the exponential distribution with density $g(x) = \exp(-(x - M))$ as the importance function.

2. Take task 1 from Section 6.8 and apply conjugate prior sampling for the error variance σ^2. Compare the posterior distributions obtained with a fixed σ^2 and by letting σ^2 vary.

8 Dynamical State Estimation

Besides model parameter estimation, where the goal is to estimate static parameters θ, one is often interested in estimating the dynamically changing *state* of the system. For instance, in our chemical reaction examples, the model state is the vector of concentrations for different compounds. In many problems, the state of the system is not known and can be observed only partially. As time proceeds, new measurements become available, that can be used to update the state estimates.

Here the state estimation problem is formulated as follows. At discrete times k, estimate the system state $\mathbf{x}_k$ using previous observations $\mathbf{y}_{1:k} = (\mathbf{y}_1, ..., \mathbf{y}_k)$, when the model and observation equations are given as

$$\mathbf{x}_k = \mathcal{M}(\mathbf{x}_{k-1}) + \varepsilon_k^p \quad (68)$$

$$\mathbf{y}_k = \mathcal{K}(\mathbf{x}_k) + \varepsilon_k^o. \quad (69)$$

In the above system, $\mathcal{M}$ is the *evolution model* that evolves the state in time and $\mathcal{K}$ is the *observation model* that maps the state to the observations. Error terms ε_k^p and ε_k^o represent the model error and the observation error, respectively.

In dynamical state estimation problems, measurements are obtained in real-time and the state estimate needs to be updated after the measurements are obtained. This can be achieved by applying the Bayes' formula sequentially. The prior is given by evolving the posterior of the model state from the previous time step using the model $\mathcal{M}$ (prediction step). The obtained prior is updated with the likelihood of the measurement (update step) to get the posterior, which is evolved with the model and used as the prior in the next time step. Repeating this procedure allows 'on-line' estimation of model states.

Dynamical state estimation techniques are needed in many important applications in various different fields. Examples of applications include, for example

- Target tracking: estimate the position and velocity of an object using a model for the dynamics and (possibly indirect) observations of the target. For instance, the Global Positioning System (GPS) uses state estimation techniques (extended Kalman filtering).
- Combining accelerometer and gyroscope data to compute the orientation, position and velocity of an object (such as a gaming device).
- Non-stationary inverse problems, for instance process tomography: estimate the dynamically changing concentrations of different compounds in a pipe by combining fluid dynamics and chemistry models with tomographic measurements [Seppänen 2009].
- Numerical Weather Prediction (NWP): estimate the state of the current weather by correcting the previous prediction with different kinds of observations (ground based, satellite etc.) to allow real-time weather predictions.
- ...

8.1 General Formulas

In general terms, the sequential state estimation, also known as filtering, works as follows. The filtering methods aim at estimating the marginal distribution of the states $p(\mathbf{x}_k|\mathbf{y}_{1:k})$ given the measurements obtained until the current time. In the prediction step, the whole distribution of the state is moved with the dynamical model to the next time step:

$$p(\mathbf{x}_k|\mathbf{y}_{1:k-1}) = \int p(\mathbf{x}_k|\mathbf{x}_{k-1})p(\mathbf{x}_{k-1}|\mathbf{y}_{1:k-1})d\mathbf{x}_{k-1}. \tag{70}$$

When the new observation $\mathbf{y}_k$ is obtained, the model state is updated using the Bayes' rule, with (70) as the prior:

$$p(\mathbf{x}_k|\mathbf{y}_{1:k}) \propto p(\mathbf{y}_k|\mathbf{x}_k)p(\mathbf{x}_k|\mathbf{y}_{1:k-1}). \tag{71}$$

This posterior is used inside integral (70) in the next prediction step. The term $p(\mathbf{x}_k|\mathbf{x}_{k-1})$ includes the evolution model and describes the probability of having state $\mathbf{x}_k$ at time k, given that the state was $\mathbf{x}_{k-1}$ at the previous time step. The idea of sequential state estimation is illustrated in Figure 21.

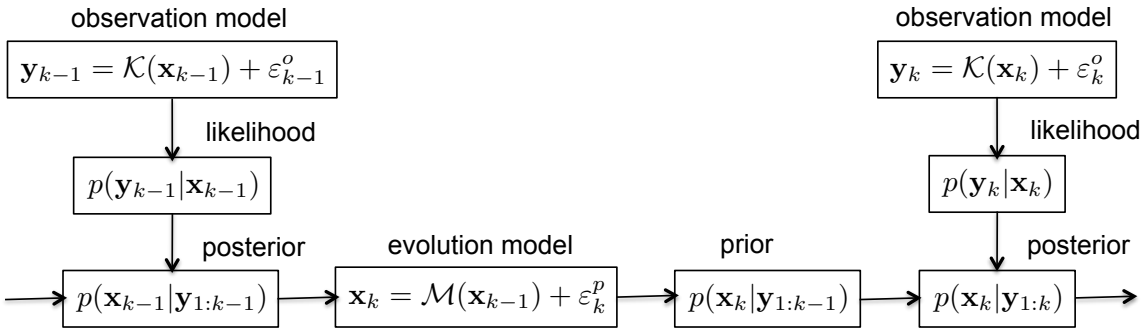

Figure 21: Two iterations of sequential state estimation at times $k-1$ and k. The prior is given by the model, which is combined with the observation to get the posterior. The posterior is further propagated to be the prior for the next time point.

Different state estimation methods can be derived, depending on the assumptions of the form of the state distribution, the likelihood and the techniques used to evolve the uncertainty of the state in time. Some of the most common methods are introduced in what follows.

8.2 Kalman Filter and Extended Kalman Filter

The Kalman filter (KF) is meant for situations, where the model is linear and the model and observation errors are assumed to be zero mean Gaussians with given covariance matrices. The extended Kalman filter (EKF) is its extension to nonlinear situations, where the model is linearized and the KF formulas are applied.

Linear Least Squares with Gaussian Prior and Gaussian Likelihood

To derive the KF formulas, let us first consider the linear model $\mathbf{y} = \mathbf{A}\mathbf{x}$ with Gaussian prior and Gaussian likelihood. Note that now the unknown is denoted by $\mathbf{x}$ instead of θ. The posterior distribution is $\pi(\mathbf{x}|\mathbf{y}) \propto p(\mathbf{y}|\mathbf{x})p(\mathbf{x})$, where the likelihood and prior are

$$p(\mathbf{x}) \propto \exp\left(-\frac{1}{2}(\mathbf{x}-\mathbf{x}_p)^T\mathbf{P}^{-1}(\mathbf{x}-\mathbf{x}_p)\right) \tag{72}$$

$$p(\mathbf{y}|\mathbf{x}) \propto \exp\left(-\frac{1}{2}(\mathbf{y}-\mathbf{A}\mathbf{x})^T\mathbf{R}^{-1}(\mathbf{y}-\mathbf{A}\mathbf{x})\right). \tag{73}$$

Here $\mathbf{x}_p$ is the prior mean and $\mathbf{P}$ and $\mathbf{R}$ are model error and measurement error covariance matrices, respectively. Maximizing the posterior density is equivalent to minimizing the negative log-likelihood, which is (ignoring the normalizing constants)

$$-2\log(\pi(\mathbf{x}|\mathbf{y})) = (\mathbf{y}-\mathbf{A}\mathbf{x})^T\mathbf{R}^{-1}(\mathbf{y}-\mathbf{A}\mathbf{x}) + (\mathbf{x}-\mathbf{x}_p)^T\mathbf{P}^{-1}(\mathbf{x}-\mathbf{x}_p). \tag{74}$$

Previously, we have derived the LSQ estimate and its covariance matrix for linear systems without the prior term $(\mathbf{x}-\mathbf{x}_p)^T\mathbf{P}^{-1}(\mathbf{x}-\mathbf{x}_p)$. Here, we attempt to do the same for the above more general expression.

Let us assume that the inverses of the covariance matrices can be symmetrically decomposed into $\mathbf{P}^{-1} = \mathbf{K}^T\mathbf{K}$ and $\mathbf{R}^{-1} = \mathbf{L}^T\mathbf{L}$ (for instance, using the Cholesky decomposition). Note that such decomposition can always be done, since covariance matrices (and their inverses) are, by definition, positive definite. Then, we can write the least squares expression as

$$-2\log(\pi(\mathbf{x}|\mathbf{y})) = (\mathbf{y}-\mathbf{A}\mathbf{x})^T\mathbf{L}^T\mathbf{L}(\mathbf{y}-\mathbf{A}\mathbf{x}) + (\mathbf{x}-\mathbf{x}_p)^T\mathbf{K}^T\mathbf{K}(\mathbf{x}-\mathbf{x}_p) \tag{75}$$

$$= (\mathbf{L}\mathbf{y}-\mathbf{L}\mathbf{A}\mathbf{x})^T(\mathbf{L}\mathbf{y}-\mathbf{L}\mathbf{A}\mathbf{x}) + (\mathbf{K}\mathbf{x}-\mathbf{K}\mathbf{x}_p)^T(\mathbf{K}\mathbf{x}-\mathbf{K}\mathbf{x}_p). \tag{76}$$

We continue by combining both terms into one expression, and write

$$-2\log(\pi(\mathbf{x}|\mathbf{y})) = (\tilde{\mathbf{y}}-\tilde{\mathbf{A}}\mathbf{x})^T(\tilde{\mathbf{y}}-\tilde{\mathbf{A}}\mathbf{x}), \tag{77}$$

where

$$\tilde{\mathbf{y}} = \begin{pmatrix} \mathbf{L}\mathbf{y} \\ \mathbf{K}\mathbf{x}_p \end{pmatrix} \quad \tilde{\mathbf{A}} = \begin{pmatrix} \mathbf{L}\mathbf{A} \\ \mathbf{K} \end{pmatrix}. \tag{78}$$

That is, we have transformed the problem into a least squares problem with identity $\mathbf{I}$ as the error covariance matrix. This can be solved with the formulas developed in Section 3: $\hat{\mathbf{x}} = (\tilde{\mathbf{A}}^T\tilde{\mathbf{A}})^{-1}\tilde{\mathbf{A}}^T\tilde{\mathbf{y}}$ and $\mathrm{cov}(\hat{\mathbf{x}}) = (\tilde{\mathbf{A}}^T\tilde{\mathbf{A}})^{-1}$. Switching back to the original notation, the terms $\tilde{\mathbf{X}}^T\tilde{\mathbf{X}}$ and $\tilde{\mathbf{X}}^T\tilde{\mathbf{y}}$ in the above formulas become

$$\tilde{\mathbf{A}}^T\tilde{\mathbf{A}} = \mathbf{A}^T\mathbf{L}^T\mathbf{L}\mathbf{A} + \mathbf{K}^T\mathbf{K} = \mathbf{A}^T\mathbf{R}^{-1}\mathbf{A} + \mathbf{P}^{-1} \tag{79}$$

$$\tilde{\mathbf{A}}^T\tilde{\mathbf{y}} = \mathbf{A}^T\mathbf{L}^T\mathbf{L}\mathbf{y} + \mathbf{K}^T\mathbf{K}\mathbf{x}_p = \mathbf{A}^T\mathbf{R}^{-1}\mathbf{y} + \mathbf{P}^{-1}\mathbf{x}_p. \tag{80}$$

That is, the LSQ solution $\hat{\theta}$ and its covariance matrix, denoted here by $\mathbf{C}$, have the expressions

$$\hat{\mathbf{x}} = (\mathbf{A}^T\mathbf{R}^{-1}\mathbf{A} + \mathbf{P}^{-1})^{-1}(\mathbf{A}^T\mathbf{R}^{-1}\mathbf{y} + \mathbf{P}^{-1}\mathbf{x}_p) \quad (81)$$

$$\mathbf{C} = (\mathbf{A}^T\mathbf{R}^{-1}\mathbf{A} + \mathbf{P}^{-1})^{-1}. \quad (82)$$

These formulas are very similar to the previously obtained linear least squares formulas (28) and (29), but here we have the prior center point $\mathbf{x}_p$ and covariance matrix $\mathbf{P}$ included.

The Kalman Filter

Let us now consider the state space model given in equations (68-69). Let us assume that both the evolution and observation models are linear, given at time step k as matrices $\mathbf{M}_k$ and $\mathbf{K}_k$, and write

$$\mathbf{x}_k = \mathbf{M}_k\mathbf{x}_{k-1} + \varepsilon_k^p \quad (83)$$

$$\mathbf{y}_k = \mathbf{K}_k\mathbf{x}_k + \varepsilon_k^o. \quad (84)$$

The model error and observation error are assumed to be zero mean Gaussian with covariance matrices $\mathbf{Q}_k$ and $\mathbf{R}_k$, respectively.

The idea of filtering is essentially to estimate the state vector $\mathbf{x}_k$ for time steps $k = 1, 2....$ In practice this is done by applying the Bayes' rule sequentially so that the prediction from the previous time step is considered as the prior, which is updated with the new measurements that become available.

To be more precise, let us assume that we have at time step $k-1$ obtained a state estimate $\mathbf{x}_{k-1}^{\text{est}}$ with covariance matrix $\mathbf{C}_{k-1}^{\text{est}}$. The prior center point for the next time step k is given by the model prediction:

$$\mathbf{x}_k^p = \mathbf{M}_k\mathbf{x}_{k-1}^{\text{est}}. \quad (85)$$

The covariance of the prediction (prior) is computed using the assumption that the state vector and model error are statistically independent:

$$\mathbf{C}_k^p = \text{Cov}(\mathbf{M}_k\mathbf{x}_{k-1}^{\text{est}} + \varepsilon_k^p) = \mathbf{M}_k^T\mathbf{C}_{k-1}^{\text{est}}\mathbf{M}_k + \mathbf{Q}_k. \quad (86)$$

This Gaussian with mean $\mathbf{x}_k^p$ and covariance matrix $\mathbf{C}_k^p$ is used as a prior, which is updated with the new measurement vector $\mathbf{y}_k$. The negative log-posterior distribution for the state vector at time step k can be written as

$$-2\log(\pi(\mathbf{x}_k)) = (\mathbf{x}_k - \mathbf{x}_k^p)^T(\mathbf{C}_k^p)^{-1}(\mathbf{x}_k - \mathbf{x}_k^p) + (\mathbf{y}_k - \mathbf{K}_k\mathbf{x}_k)^T\mathbf{R}_k^{-1}(\mathbf{y}_k - \mathbf{K}_k\mathbf{x}_k), \quad (87)$$

which is just the linear least squares problem with Gaussian prior and Gaussian likelihood discussed in the previous section. That is, for time step k, the estimate and its covariance can be computed using the formulas (81-82).

However, for computational reasons, the Kalman filter formulas are usually written in the following form, which can be obtained via direct but somewhat non-trivial matrix manipulations (see exercises):

$$\mathbf{G}_k = \mathbf{C}_k^p\mathbf{K}_k^{\text{T}}(\mathbf{K}_k\mathbf{C}_k^p\mathbf{K}_k^{\text{T}} + \mathbf{R}_k)^{-1} \quad (88)$$

$$\mathbf{x}_k^{\text{est}} = \mathbf{x}_k^p + \mathbf{G}_k(\mathbf{y}_k - \mathbf{K}_k\mathbf{x}_k^p) \quad (89)$$

$$\mathbf{C}_k^{\text{est}} = \mathbf{C}_k^p - \mathbf{G}_k\mathbf{K}_k\mathbf{C}_k^p. \quad (90)$$

In the above formulas, $\mathbf{G}_k$ is called the *Kalman gain* matrix. To sum up, we can write the Kalman filter as an algorithm as follows:

1. **Prediction**: move the state estimate $\mathbf{x}_{k-1}^{\text{est}}$ and its covariance $\mathbf{C}_{k-1}^{\text{est}}$ in time
   (a) Compute $\mathbf{x}_k^p = \mathbf{M}_k \mathbf{x}_{k-1}^{\text{est}}$.
   (b) Compute $\mathbf{C}_k^p = \mathbf{M}_k \mathbf{C}_{k-1}^{\text{est}} \mathbf{M}_k^{\text{T}} + \mathbf{Q}_k$.
2. **Update**: combine the prior $\mathbf{x}_k^p$ with observations $\mathbf{y}_k$
   (a) Compute the Kalman gain $\mathbf{G}_k = \mathbf{C}_k^p \mathbf{K}_k^{\text{T}} (\mathbf{K}_k \mathbf{C}_k^p \mathbf{K}_k^{\text{T}} + \mathbf{R}_k)^{-1}$.
   (b) Compute the state estimate $\mathbf{x}_k^{\text{est}} = \mathbf{x}_k^p + \mathbf{G}_k(\mathbf{y}_k - \mathbf{K}_k \mathbf{x}_k^p)$.
   (c) Compute the covariance estimate $\mathbf{C}_k^{\text{est}} = \mathbf{C}_k^p - \mathbf{G}_k \mathbf{K}_k \mathbf{C}_k^p$.
3. Increase k and go to step 1.

The Extended Kalman Filter

The extended Kalman filter (EKF) is an extension of the Kalman filter to the case where the evolution and/or observation model are nonlinear. The EKF directly uses the Kalman filter formulas in the nonlinear case by replacing the nonlinear model and observation operators in the covariance computations with linearizations: $\mathbf{M}_k = \partial \mathcal{M}(\mathbf{x}_{k-1}^{\text{est}})/\partial \mathbf{x}$ and $\mathbf{K}_k = \partial \mathcal{K}(\mathbf{x}_k^p)/\partial \mathbf{x}$. For small scale models, the linearizations can be computed numerically, using finite differences.

8.3 Ensemble Kalman Filtering

In ensemble filtering, the uncertainty in the state estimate $\mathbf{x}_k$ is represented as N samples instead of a covariance matrix, here denoted as $\mathbf{s}_k = (\mathbf{s}_{k,1}, \mathbf{s}_{k,2}, ..., \mathbf{s}_{k,N})$. The first ensemble filtering method was the ensemble Kalman filter (EnKF), see, for instance, [Evensen 2004]. The EnKF essentially replaces the state covariance matrices in EKF with the sample covariance calculated from the ensemble. The sample covariance can be written as $\text{Cov}(\mathbf{s}_k) = \mathbf{X}_k \mathbf{X}_k^{\text{T}}$, where

$$\mathbf{X}_k = ((\mathbf{s}_{k,1} - \overline{\mathbf{s}_k}), (\mathbf{s}_{k,2} - \overline{\mathbf{s}_k}), ..., (\mathbf{s}_{k,N} - \overline{\mathbf{s}_k}))\, /\sqrt{N-1}. \tag{91}$$

The sample mean is denoted by $\overline{\mathbf{s}_k}$. Using this notation, the EnKF algorithm can be formulated as follows:

1. **Prediction**: move the state estimate and covariance in time
   (a) Move ensemble forward and perturb members with model error: $\mathbf{s}_{k,i}^p = \mathcal{M}(\mathbf{s}_{(k-1),i}^{\text{est}}) + \mathbf{e}_{k,i}^p$, $i = 1, ..., N$.
   (b) Calculate sample mean $\overline{\mathbf{s}_k^p}$ and covariance $\mathbf{C}_k^p = \mathbf{X}_k \mathbf{X}_k^{\text{T}}$.
2. **Update**: combine the prior with observations
   (a) Compute the Kalman gain $\mathbf{G}_k = \mathbf{C}_k^p \mathbf{K}_k^{\text{T}} (\mathbf{K}_k \mathbf{C}_k^p \mathbf{K}_k^{\text{T}} + \mathbf{C}_{\varepsilon_k^o})^{-1}$.

(b) Update ensemble members $\mathbf{s}_{k,i}^{\text{est}} = \mathbf{s}_{k,i}^{p} + \mathbf{G}_k(\mathbf{y}_k - \mathbf{K}_k\mathbf{s}_{k,i}^{p} + \mathbf{e}_{k,i}^{o})$.

(c) Calculate state estimate as the sample mean: $\mathbf{x}_k^{\text{est}} = \overline{\mathbf{s}_k^{\text{est}}}$.

3. Increase k and go to step 1.

In the above algorithm, vectors $\mathbf{e}_{k,i}^{p}$ and $\mathbf{e}_{k,i}^{o}$ are realizations of the model and observation errors ε_k^{p} and ε_k^{o}, respectively.

The ensemble Kalman Filter is simple to implement and it does not require linearization of the forward model. In addition, the computations are trivially parallelizable, since the N forward model evaluations can be run in parallel. Lately, ensemble Kalman filtering has become an active research topic in large scale filtering problems, such as atmospheric data assimilation, and numerous variants of the basic EnKF scheme have been developed.

8.4 Particle Filtering

The Kalman filtering methods described in the above sections assume a Gaussian form for the filtering distributions. Particle filtering methods are fully statistical state estimation methods in the sense that they do not rely on any assumptions about the form of the target, and all inference is carried out by Monte Carlo sampling. Particle filtering methods, often also called *sequential Monte Carlo* (SMC) methods, are based on sequential application of the importance sampling method described in Section 7.5.

Let us describe the state estimate at time $k-1$ with samples $\mathbf{s}_{(k-1,i)}^{\text{est}}$, where $i = 1, ..., N$. In particle filtering, the forward model $p(\mathbf{x}_k|\mathbf{x}_{k-1})$ is used to move the particles forward to obtain the predicted particles, or prior particles $\mathbf{s}_{k,i}^{\text{p}}$. The predicted particles are considered to be samples from the prior distribution at time step k, and the prior distribution is considered as the importance function for sampling from the posterior. That is, the importance weight for particle i at time k becomes

$$w_{k,i} = \frac{\pi(\mathbf{s}_{k,i}^{\text{p}}|\mathbf{y}_k)}{p(\mathbf{s}_{k,i}^{\text{p}})} = \frac{p(\mathbf{y}_k|\mathbf{s}_{k,i}^{\text{p}})p(\mathbf{s}_{k,i}^{\text{p}})}{p(\mathbf{s}_{k,i}^{\text{p}})} = p(\mathbf{y}_k|\mathbf{s}_{k,i}^{\text{p}}). \tag{92}$$

That is, the importance weights for the particles can be computed directly using the likelihood function. Intuitively, predicted particles that hit closer to the next observation get a larger weight than points that do not predict the observation that well.

Once the importance weights are obtained, the posterior particles $\mathbf{s}_{k,i}^{\text{est}}$ are sampled with replacement from the prior particles in proportion to the importance weights. That is, the prior particles that match the observations well are repeated many times in the posterior particles, and the less likely particles are discarded.

The particle filter implemented in this way is called the *Sequential Importance Resampling* (SIR) algorithm. The SIR method can be written as an algorithm as follows:

1. Move the particles forward: $\mathbf{s}_{k,i}^{\text{p}} \sim p(\mathbf{s}_{k,i}|\mathbf{s}_{(k-1,i)}^{\text{est}})$, with $i = 1, ..., N$.

2. Compute the importance weights $w_{k,i} = p(\mathbf{y}_k|\mathbf{s}_{k,i}^{\text{p}})$, with $i = 1, ..., N$

3. Resample with replacement in proportion to the importance weights to obtain the posterior particles $\mathbf{s}_{k,i}^{\text{est}}$.

4. Increase k and go to step 1.

Note that the particle filter is here described using general forms for the evolution and observation models, and not particularly for the additive state space model given in equations (68-69). This is the way particle filtering is usually written in the literature, and it emphasizes that no assumptions of the forms of the densities are assumed. Naturally, the additive model can be used here as well: then, in step 1 above, the particle evolution would be computed as in the EnKF, $\mathbf{s}_{k,i}^{p} = \mathcal{M}(\mathbf{s}_{(k-1),i}^{\text{est}}) + \mathbf{e}_{k,i}^{p}$ where $\mathbf{e}_{k,i}^{p}$ are realizations of the model error.

While particle filtering does not contain simplifying assumptions of the underlying densities, and therefore in principle gives correct results as the number of particles N increases, it has its problems and it can be cumbersome to use it in practice. For instance, the number of particles required to get accurate estimates grows fast as a function of the dimension of the state vector. Therefore, the particle filtering method is not very attractive in high-dimensional problems, and the methods based on, e.g., Gaussian approximations, are often more effective. Another known problem in particle filtering is the possible degeneration of the importance weights: there is a risk that all the weight is put into one sample, which can yield poor filtering results. Many strategies have been developed to overcome these issues. Currently, however, the applicability of particle filtering is limited to small dimensional state estimation problems.

8.5 Exercises

1. An object is moving on (x_1, x_2) plane. We have noisy observations of the position of the object at different time points. The observations are obtained with time interval $\Delta t = 1$. The goal is to estimate the location and the velocity with the Kalman Filter.

   (a) Write the model and observation equations by assuming that the velocity is constant within each time interval.

   (b) Implement the Kalman Filter using the measurements given in the file `observations.dat`. Use $(x_1, x_2) = (10, 10)$ and velocity $(1, 0)$ as the initial point. Test the effect of different scales for the model error covariance matrix and for the measurement error covariance matrix.

2. The famous Lorenz equation ODE system

$$\begin{aligned} \frac{dx}{dt} &= a(y - x) \\ \frac{dy}{dt} &= x(b - z) - y \\ \frac{dz}{dt} &= xy - cz \end{aligned}$$

is a simple prototype of chaotic dynamics (that is seen, for instance, in weather systems). In the file `lor3data.dat`, observations for the components x and y are given. The file is organized so that the first column gives the measurement time instances and the following columns are noisy observations for x and y.

Choose a filtering method and implement it to estimate the states (x, y, z) at different time points. Use initial values $x(0) = y(0) = z(0) = 1$ and parameter values $a = 10$, $b = 28$, $c = 8/3$.

To see how accurate your filter is, compare the state estimates to the true state values given in the file `lor3truth.dat`. The first column of the file gives the time instances and the following columns are the true values for x, y and z used to generate the data.

In your filter, use the true measurement error variance $\sigma^2 = 0.1\mathbf{I}$ that was used to generate the observations. Study different scales for the model error covariance matrix and see how it affects the accuracy of the filter. In addition, if you choose a Monte Carlo filtering method, study how the number of ensemble members (or particles) affects the accuracy.

3. Starting from the least squares formulas (81-82), derive the Kalman filter formulas (88-90). Hint: use the Sherman-Morrison-Woodbury matrix inversion lemma (check e.g. Wikipedia).

References

[Brooks et al. 2011] Brooks S., Gelman A., Jones G.L., Meng X., 2011. Handbook of Markov Chain Monte Carlo. Chapman & Hall / CRC, USA.

[Evensen 2004] Evensen G., 2007. Data assimilation: The ensemble Kalman filter. Springer, Berlin.

[Forsythe et al. 1977] Forsythe G.E., Malcom M.A., Moler C.B., 2009. Computer Methods for Mathematical Computations. Prentice-Hall series in automatic computation.

[Gelman et al. 1996] A. Gelman, G. O. Roberts, and W. R. Gilks. Efficient Metropolis jumping rules. Bayesian Statistics, 5, 599607, 1996.

[Gelman et al. 1996] Gelman A., Carlin J., Stern H. and Rubin D., 2004. Bayesian Data Analysis, Second Edition. London, Great Britain: Chapman Hall.

[Haario et al. 2001] Haario H., Saksman E. and Tamminen J., 2001. An Adaptive Metropolis Algorithm. Bernoulli, 7(2), pages 223-242.

[Haario et al. 2005] Haario H., Saksman E. and Tamminen J., 2005. Componentwise adaptation for high dimensional MCMC. Comput. Stat., 20(2), pages 265273.

[Haario et al. 2006] Haario H., Laine M, Mira A., Saksman E., 2006. DRAM: Efficient adaptive MCMC. Stat. Comput., 16, pages 339–354.

[Hastings 1970] Hastings, W.K. (1970). Monte Carlo Sampling Methods Using Markov Chains and Their Applications. Biometrika 57 (1): 97109.

[Laine 2008] Laine M., 2008. Adaptive MCMC Methods with Applications in Environmental and Geophysical Models. Finnish Meteorological Institute Contributions, 69.

[Metropolis et al. 1953] Metropolis N., Rosenbluth A.W., Rosenbluth M.N., Teller A.H. and Teller E., 1953. Equations of State Calculations by Fast Computing Machines. Journal of Chemical Physics, 21(6), pages 1087–1092.

[Mira 2001] Mira A., 2001. On Metropolis-Hastings algorithms with delayed rejection. Metron, LIX(34), pages 231–241.

[Seppänen 2009] Seppänen A., Voutilainen A., Kaipio J.P. State estimation in process tomography – reconstruction of velocity fields using EIT. Inverse Problems, 25(8), 2009.