Stochastic Simulation and Monte Carlo Methods

Andreas Hellander

March 31, 2009

1 Stochastic models, Stochastic methods

In these lecture notes we will work through three different computational problems from different application areas. We will simulate the irregular motion of a particle in an environment of smaller solvent molecules, we will study a biochemical control network and we will compute a numerical approximation to a high dimensional integral. Even though these three problems are different, and the way we will solve them will be different, they have one thing in common. In each case we will use a *Monte Carlo* method.

There is not a single definition of a Monte Carlo method, but they have in common that they make use of random sampling to compute the result. The algorithms typically rely on *pseudo random numbers*, computer generated numbers mimicking true random numbers, to generate a *realization*, one possible outcome of a process. All outcomes do not have to be equally probable, and by repeating the procedure with different random numbers as input, one gathers data corresponding to the modeled process. On this data, one may then perform a statistical analysis in order to answer different questions about the process. We illustrate the general principle in the following example, which we will study in more detail using a computer in Exercise 8.

1. We have a model of some physical phenomena and some question regarding the system we are modeling we like to answer. For example, suppose that we model the motion of a single large protein in a surrounding of water molecules and that the system is confined to a box of some fixed size. Suppose we want to know the *average time* until the particle hits one of the sides of the box if it starts in the center.

2. The model is stochastic by construction, and one possible path of the particle can be generated by a simple algorithm that repeatedly gen-

erates pseudo random numbers from the standard normal distribution. This means that if we generate a number of paths, they will all be different. In order to answer the question, we use the simple algorithm to generate many different realizations, say 10^5, and for each realization we measure and store the time it took until the particle hit one of the sides.

3. Once we have gathered the data set, the different hitting times in this case, we simply compute the arithmetic mean in order to get an approximation of the average time.

In the following, we will make a clear distinction between stochastic models and stochastic methods. For our particle example above, the model is stochastic, but in order to answer the question regarding the hitting time we could have used a deterministic method. We can formulate a partial differential equation (PDE), which if we solve it gives precisely the average time until the particle leaves the box. If we used this approach, the method would no longer be a Monte Carlo method, but the model would still be stochastic. However, it may be much harder to solve the PDE than to repeatedly sample the process in a Monte Carlo method, in particular if the motion takes place in higher dimensions.

A classical use of a Monte Carlo method to solve a *deterministic problem* is to evaluate definite integrals. Even in this case, the methodology follows the same basic principle as in the above example. Here, the 'model' is an integral, and it is solved by repeatedly 'sampling' the integrand assuming that the independent variable is distributed according to some probability distribution. In this way, we collect data and by identifying the integral with a certain expected value we can approximate it by computing the arithmetic mean of the data. The procedure is summarized below and why this method works will be explained in Section 3. Note the similarity with the three steps in the particle motion problem above.

1. Suppose we want to compute the value of the integral $I = \int_{\Omega} f(x)dx$ over some domain Ω.

2. We generate a data set made up of the function f evaluated at random points x distributed uniformly in Ω. These points are generated using a pseudo random number generator.

3. We compute an approximation to I by computing the arithmetic mean of the data gathered in the previous step and multiplying the result with the volume of Ω.

In this case, the underlying problem is deterministic, and the use of Monte Carlo to solve it is motivated by the fact that for very high dimensions, especially for non-smooth integrands, it is more *computationally efficient* to solve it with MC than using a traditional quadrature rule (remember Scientific Computing I). In this example, we use a *stochastic method* to solve a *deterministic problem* for efficiency reasons.

In summary, Monte Carlo methods can be used to study both deterministic and stochastic problems. For a stochastic model, it is often natural and easy to come up with a stochastic simulation strategy due to the stochastic nature of the model, but depending on the question asked a deterministic method may be used. The use of a stochastic method is often motivated by the fact that a deterministic method to answer the same question is not available, that it is is too complicated to be practically useful, or that it is computationally intractable, which is often the case if the problem is high dimensional. On the other hand, when a deterministic method is applicable, it is often preferable due to the very slow convergence of Monte Carlo methods.

No matter the reason for using Monte Carlo methods, they will inevitably require many random numbers. The quality of the pseudo random generators is crucial to the correctness of the results computed with a stochastic algorithm, and the speed with which they are generated is vital to the performance of the stochastic method. There are many different algorithms in use to generate pseudo random numbers, and the development of even more efficient algorithms is an active field of research. Many courses on computational statistics and Monte Carlo discusses the most basic algorithms. We will not do this here, since we will focus on Monte Carlo methodology at a more basic level. However, all major programming languages have some routine for generating pseudo random numbers from (at least) the uniform distribution, and there are many other generators freely available for non commercial use. In the sections that follow, we will see examples of how we can use such an 'black box' routine to obtain random numbers from other distributions that we will need.

2 Two important theorems

In the following we will assume familiarity with standard statistical concepts such as expected value, variance and probability distribution. These notions may be reviewed in the course book [1]. In addition, Monte Carlo relies heavily on two famous theorems from probability theory, namely the *law of large numbers (LLN)* and the *central limit theorem (CLS)* which we will state

without proof. If you are familiar with these theorems, you may skip this section. They will not be explicitly needed in order to work through the examples in the following sections, but they are required in order to really understand why MC works. If it seems artificial to you at this time, you may want to come back to this section after reading the case studies.

Every time we estimate some unknown quantity by generating a sample (the data set composed of the realizations we generate) and computing the sample average as we will do when we evaluate integrals with Monte Carlo, we rely on the law of large numbers. Even if you have not seen the theorem before, you have probably used it many times, especially if you have taken some experimental courses. It states that if we repeatedly sample a stochastic variable, the sample mean converges to the true expected value. Loosely speaking, this result says that the Monte Carlo method is *consistent*.

Theorem 1 (weak) Law of large numbers (LLN)

Let $X_1, X_2, \ldots, X_n$ be independent random variables with mean μ and set $\bar{X}_n = \frac{1}{n}\sum_{i=1}^{n} X_i$. Then $P(\bar{X}_n - \delta < \mu < \bar{X}_n + \delta) \to 1$ as $n \to \infty$ for any $\delta > 0$.

Computer exercise 1

Implement a small `Matlab` routine that simulates throws with a dice. For each successive throw, compute the arithmetic mean based on all throws so far and plot it as a function of N, the number of throws. To which value does the mean appear to converge?

The LLN provides us with consistency, but does not give any information regarding the *error* in the approximation or the *rate of convergence* to the true expected value. This is taken care of by the central limit theorem.

Theorem 2 Central limit theorem (CLS)

Let $X_1, X_2, \ldots, X_n$ be a sequence of independent identically distributed random variables with finite mean μ and variance $\sigma^2 > 0$ and let $\bar{X}_n = \frac{1}{n}\sum_{i=1}^{n} X_i$. Then $\sqrt{n}(\bar{X}_n - \mu) \xrightarrow{D} N(0, \sigma^2)$ as $n \to \infty$

The CLS states that the distribution of the error in the sample mean μ tends to a normal distribution and that the variance decay as N^{-1} if N is the sample size and this means that the error decays as $N^{-1/2}$. Thus, Monte Carlo methods are converging very slowly, and are often computationally intensive. However, the convergence rate is insensitive to the dimensionality

of the problem, which is generally not the case for deterministic methods.

Computer exercise 2

Use the code from the previous exercise and for a fixed number of throws N, simulate the dice and compute the arithmetic mean. Repeat this $M = 100$ times and store the means in an array. Vary the value of N in the range $100 - 100.000$ and for each N plot the mean values in a histogram plot using the built in routine `hist`.

3 Approximating a high dimensional integral using Monte Carlo

As we saw in Computer exercise 2, the arithmetic mean based on N samples took a slightly different value every time we repeated the experiment. The computed value of the mean is a *random variable*, meaning that it is not determined deterministically. Instead, we noticed that if we plotted a histogram when we repeated the experiment, the histogram had the shape of the normal distribution (this is the content of the CLT). The distribution determines the random variable, as it gives information of the *probability* that the arithmetic mean takes a value in a specific range. The probability is higher for taking a value close to the expected value (true mean) and it decays as we move further away from the mean. In this case, we say that the arithmetic mean is (assymptotically) a *normally distributed random variable* and we write $\bar{\mu} \sim \mathcal{N}(\mu, \sigma^2)$. The normal distribution has the form

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{1}$$

Obviously, not all random variables are normally distributed, but in the same way every (continuous) random variable X has an associated probability distribution, $f(x)$. It defines the random variable by providing information of the probability that a realization falls in a domain Ω (a measurable set). The situation is similar for discrete random variables, but we will only discuss the continuous case in this section. For example,

$$P(X \in [a, b]) = \int_a^b f(x)dx$$

gives the probability that the one dimensional random variable X takes a

value in the interval $[a, b]$. The *expected value*, or mean value, of a (continuous) random variable X is defined by

$$E[X] = \int_{-\infty}^{\infty} x f(x) dx. \tag{2}$$

Similarly, if $g(X)$ is an integrable function of the random variable,

$$E[g(X)] = \int_{-\infty}^{\infty} g(x) f(x) dx \tag{3}$$

is the mean value of g provided X is distributed according to $f(x)$. We will use this to compute the integral

$$I = \int_a^b g(x) dx. \tag{4}$$

For sake of clarity, we present the theory in only one dimension. The extension to higher dimensions will be done in Computer exercise 4. A random variable uniformly distributed in the interval $[a, b]$ has equal probability of falling in any subinterval of equal size. Its density is given by

$$f(x) = \frac{1}{b-a}, \quad x \in [a, b] \tag{5}$$

so assuming that $g(X)$ is a function of a uniformly distributed random variable, its expected value is given by

$$E[g(X)] = \int_a^b g(x) \frac{1}{b-a} dx = \frac{I}{b-a}. \tag{6}$$

Relying on the law of large numbers, we can compute an approximation to I by sampling pseudo random numbers from the standard uniform distribution, scaling them to $[a, b]$, evaluating g at these points and computing the arithmetic mean of the resulting data set

$$I = \int_a^b g(x) dx \approx \frac{b-a}{N} \sum_{i=1}^{N} g(x_i), \tag{7}$$

where N is the number of points.

Exercise 1

Consider the integral

$$I = \int_{-1}^{1} x^2 e^{\frac{-x^2}{2}} dx.$$

We would like to compute an approximation using Monte Carlo integration. Propose two possible approaches. You can assume that pseudo random number generators for different distributions are available. *Hint: Remember the normal distribution.*

Exercise 2

A classical example is how to estimate π using Monte Carlo. It can be done by sampling uniform random numbers in the unit square and counting how many of them that fall into the inscribed quadrant of the unit circle. Since we know that the area of the circle quadrant is $\pi r^2/4$ and that of the square is r^2, the fraction of the area of the quarter circle to the area of the square is $\pi/4$. At a first glance, this may not seem like a integration problem, but it can be formulated as such. Write down the (double) integral (and thus the expected value) that correspond to this procedure, i.e. the integral $I = 4E[g(X)] = \pi$. Explain how you determine if the points fall into the circle quadrant.

Computer exercise 3

Implement a routine that estimates the value of π using the procedure in Exercise 2. Try to avoid using any loops. *Hint: Consider using the built in function 'find'* .

By replicating the entire procedure involved in computing the approximation of the integral we can estimate the error in the computed value of I. If we make M independent MC estimations of I, we use the pooled mean

$$\hat{I} = \frac{1}{M} \sum_{j=1}^{M} I_j \tag{8}$$

where each I_j is computed according to (7), as the approximation of I. The error in $\hat{I}$ is bounded by

$$|\epsilon| \leq \frac{1.96\hat{s}}{\sqrt{M}} \tag{9}$$

with 95% probability. Here, $\hat{s}^2$ is the sample variance

$$\hat{s^2} = \frac{1}{M-1} \sum_{j=1}^{M} (\hat{I} - I_j)^2. \tag{10}$$

This gives us a 95% confidence interval for I

$$I \in [\hat{I} - |\epsilon|, \hat{I} + |\epsilon|]. \tag{11}$$

The confidence level can be modified by changing the α-factor (1.96 in this case), cf. Student's t-distribution or consult a basic statistics textbook for more information.

Computer exercise 4

In this exercise we will use Monte Carlo integration to compute an approximation to an integral in 10 dimensions. The integrand will simply be the density for the multivariate normal distribution

$$f(\mathbf{x}) = \frac{1}{(2\pi)^{d/2}|V|^{1/2}} e^{-\frac{1}{2}(\mathbf{x}-\mu)^T V^{-1}(\mathbf{x}-\mu)}$$

where d is the dimension, V is the $d \times d$ covariance matrix and $|V|$ is the determinant of V. For the standard normal distribution, $V = I$, the identity matrix, and $\mu = 0$. We know from the properties of probability densities that the integral over the whole space is $I = \int_{\mathbb{R}^d} f(x)dx = 1$. Obviously, we can not integrate over the whole space, so we need to make a truncation. As the domain we will take the d-dimensional hypercube $[-5, 5]^d$. Outside this domain the probability is small, and we do not loose much probability mass leaving the rest of the space out. The following `Matlab` function evaluates the standard normal distribution in arbitrarily dimensions.

```
% The density of the d-dim standard normal distribution.
%
% Andreas Hellander, 2009.
%
% Input: x - (d,N) matrix. Each column of x is a point
%             where the function is evaluated.
%

function y = fnorm(x)

   [d,N]=size(x);
   C = eye(dim);
   y = 1/(2*pi)^(d/2)*exp(-0.5*sum(x.*(C*x),1));
```

The matrix x is a $d \times N$ matrix, where each column is a point in d-dimensional space. This means that you can evaluate N such points in a single call to `fnorm`.

Write a `Matlab` routine `mcint` that evaluates an integral in d-dimensions and estimates the mean and computes a 95% confidence interval according to (11). The function should confine to the specification below

```
% MCINT. Simple routine to compute an integral using Monte Carlo.
%
%  Some Student, 2009.
%
%  Input:  N - the number of points in each realization.
%          M - the number of repetitions used for error
%              estimation. (Recommendation, M = 30+).
%          Total number of points used is thus M*N.
%
%          D - the domain. Supports only hypercubes in 'dim' dimensions.
%              Each row in the matrix D represents the range of the
%              corresponding variable. Example, the unit cube
%              would be given by
%
%              D = [0 1;
%                   0 1;
%                   0 1];
%
%          f - function y = f(x): defining the integrand. Must
%              evaluate f for a vector of dim*N points (x).
%
%

function [val,err] = mcint(f,D,N,M)
```

Use this function to estimate the value of the integral of the normal distribution in 10 dimensions. Experiment with varying N and M. Also, vary d and study the convergence rate as a function of N.

Note: In this case, the integrand is a smooth function, and Monte Carlo integration would likely be outperformed by an adaptive, deterministic quadrature. With that said, do you think it is as easy to implement such a code in arbitrary dimensions?

What we have studied in this section is the most basic form of Monte Carlo integration. In reality, it is often used in connection with some *variance reduction* method. These techniques reduce the variance in the estimated mean, and thus allow for a better result with fewer sample points N. Another

popular improvement is to use so called quasi-random sequences instead of pseudo random numbers. With this technique, it is possible to improve the convergence rate from $1/\sqrt{N}$ to almost $1/N$.

In summary, Monte Carlo integration is a relatively easy to implement method to perform numerical integration. It converges very slowly with the number of sample points N, but the convergence rate is almost independent of the dimension of the integral, and it is not sensitive to the smoothness of the integrand (this is not true for quasi-Monte Carlo). This is in contrast to deterministic quadrature rules that converge much faster, but degrades in performance as the dimension increases. Adaptive quadrature rules is possible to use in fairly high dimensions, but they often require a smooth integrand to perform well. In practical applications, Monte Carlo integration is often used together with some method to reduce the variance in the estimate. In computational financial mathematics, integrals in dimension $30 - 100$ is not uncommon and there Monte Carlo is widely used.

4 When noise matters – case studies

In the previous section we used Monte Carlo to compute a numerical approximation to an integral. The underlying problem in that section was deterministic, in the sense that the independent variable was a deterministic variable, even though we interpreted it a stochastic variable in order to identify the integral with an expected value in the Monte Carlo method. In the following sections we will look at two different examples where the underlying mathematical model is stochastic. In the first case we will study a single particle diffusing, modeled by Brownian motion. In this model, the position of the particle is a stochastic variable. In the second example, the stochastic variable will be the number of molecules of proteins in a cell, and the mathematical model is a discrete Markov process. In both cases, it will be natural to formulate a stochastic simulation strategy to study how the system behaves over time.

4.1 Free diffusion of a particle – Brownian motion

One of the most fundamental processes in models of dynamical phenomena in many application areas is diffusive mass transport. The deterministic model of this process is a *partial differential equation* (PDE) and this type of equations will be studied in detail in the course *Scientific Computing III*. An important aspect to understand right now is the difference between the deterministic model and the particle model we consider here. Slightly

simplified, the PDE describes the *average* behavior of the process, while the particle model describes the irregular motion of single particles moving in a surrounding of a large number of (smaller) particles exerting forces on the particle. A useful example is to consider a large protein moving in an aqueous solution, where the large number of water molecules "pushes" the protein in different directions due to the thermal energy in the system.

Here, we will focus on the fundamental model of the particle's movement introduced to the physics community by Albert Einstein in 1905. Commonly, the discovery of Brownian motion is considered to be due to Robert Brown in 1827 when he, using a microscope, studied the motion of pollen particles floating in water. Einstein used the process to indirectly confirm the existence of atoms and molecules in his fundamental paper 1905 and he postulated what properties the mathematical model of Brownian motion must have. From a mathematical perspective, the model of Brownian motion is one of the simplest *stochastic processes* and it was given a rigorous mathematical foundation by Norbert Wiener. Because of this, the process is also often called the *Wiener process*. Regarded as a stochastic process, it has numerous applications in both applied and pure mathematics. Brownian motion frequently appears as the noise term in many *stochastic differential equations* (SDEs) and it is therefore important that we understand how to simulate this process.

In simple words, a stochastic process is a collection of random variables. In many cases, they are indexed by time. Unlike the deterministic case, where the future behavior of a process is uniquely determined by the initial condition, a stochastic process will take different paths if simulated repeteadly. Not all paths are equally probable though, and they are governed by some probability distribution.

Here, the random variables are the position of the particle at time t. So for each time $t \geq t_0$, the position is a stochastic variable, and we can write the process $\{B_t : t \geq 0\}$. While a mathematical discussion of Brownian motion is way out of the scope of this course, we state below the properties that characterize the process B_t as it will tell us how to *simulate* it in a simple numerical algorithm.

1. $B_0 = 0$ (almost surely).
2. B_t is continuous (almost surely).
3. B_t has independent, normally distributed increments, $B_t - B_s \sim \mathcal{N}(0, t-s), \quad 0 \leq s \leq t.$

Property 1 above simply means that the process starts in origo. If we want to simulate it starting in another point, say x_0, we can simply simulate the

process $X_t = x_0 + B_t$. Property 2 is important and it is studied in some detail in higher courses. For now, however, we just take it for granted. Property 3 immediately suggests an algorithm to simulate paths of the Brownian motion. If we choose a time step Δt we know that $B_{t+\Delta t} - B_t \sim \mathcal{N}(0, \Delta t)$. Suppose also that we know the particle's position at time t, x_t, we can sample the position at $t + \Delta t$ by drawing a random number $x_{t+\Delta t}$ from the normal distribution $\mathcal{N}(x_t, \Delta t)$.

Exercise 3

Suppose that a random number generator is available that generates random numbers according to the standard normal distribution $\mathcal{N}(0, 1)$. Using this generator, explain how can you generate random numbers from the distribution $\mathcal{N}(x_t, \Delta t)$.

We can now state an algorithm to generate a realization of the process. It is given in pseudocode in Algorithm 1.

Algorithm 1 Free diffusion of a particle

Initialize: choose a time step Δt, an initial position x_0 and a final time T_f. Set $t = t_0$ and $x_{t_0} = x_0$.
while $t < T_f$ **do**
  Generate a random number $R \sim \mathcal{N}(0, 1)$.
  $x_{t+\Delta t} = x_t + \sqrt{\Delta t} R$.
  $t = t + \Delta t$.
end while

Figure 1 shows one trajectory of a freely diffusing particle starting in origo.

Computer exercise 5

Write a `Matlab` function that uses a while or for-loop in order to simulate the motion of a particle undergoing 2D Brownian motion according to Algorithm 1. The input to the function should be the final time T_f, the starting point x_0 in the plane and the time step Δt. A n–dim Brownian motion is composed of n independent 1D motions. Normally distributed random numbers can be generated in `Matlab` using the function `randn`.

Computer exercise 6

In the previous exercise we implemented Algorithm 1 using a while or for-loop. In `Matlab` this should usually be avoided for efficiency reasons. Instead

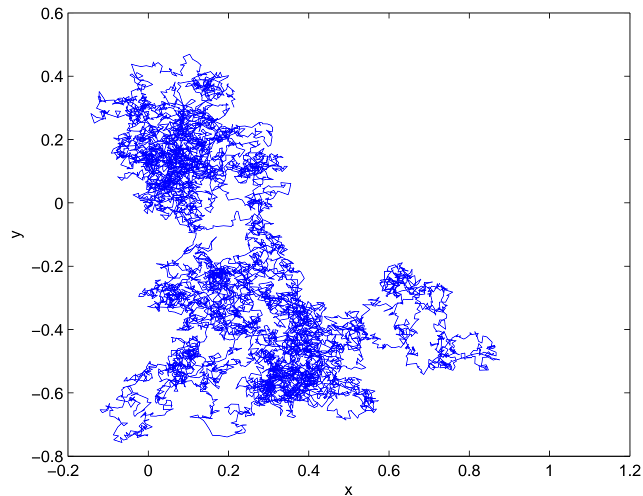


Figure 1: One path of a freely diffusing particle simulated with Algorithm 1.

one should try to use the built-in functions as much as possible. Below is a `Matlab` function that does the same thing without any loop.

```
% Brownian motion.
%
% Input:    Tf   - the final time
%           x0  - starting point (the dimension of the motions is numel(x0))
%           dt   - time step.
%
% Andreas Hellander, 2009.
%
%%
function X = brown(Tf, x0, dt)

    dim = numel(x0);
    N = floor(Tf/dt);
    X = repmat(x0,N,1);
    noise = randn(N,dim);
    X=X+sqrt(dt)*cumsum(noise);
```

Using the `help` function in `Matlab`, find out what the functions `repmat` and `cumsum` do and explain why the above implementation do the same thing as the one in the previous exercise. Note that the above implementation works

in arbitrarily dimensions.

Computer exercise 7

Implement the function in the previous exercise and, using the functions `tic` and `toc`, compare the time required to generate a realization with $T_f = 1$, $x_0 = [0, 0]$ and $\Delta t = 1e - 8$ using the implementation with a loop and the one without. Which implementation requires most memory?

Computer exercise 8

There are several interesting questions one can ask about a particle undergoing Brownian motion. One such questions is, giving that the particle starts its motion in the center of the interval (a, b), what is the average time until the particle leaves that interval, $E[\tau]$. For a 1D Brownian motion, this problem has a simple analytical solution, $E[\tau] = (b - a)^2$. We will now try to verify this using a simulation.

a.)

Modify the codes from the previous exercises (which variant is most suitable in this case?) to simulate the Brownian motion until it exits the interval (a, b) and then, return the time that passed. The function should look like

```
function exit_time = brown_exit(interval, x0, dt)
```

where the first argument is the interval given as a vector $[a, b]$ and x_0 and dt is as in the previous exercises.

b.)

Using the function from a.), write a script that computes the mean value of the exit time based on $M = 100$ realizations. Let the particle start in origo and take the interval $(-0.5, 0.5)$. Compare this value with the analytical solution.

c.)

Repeat the calculation in b.) for an increasing number of realizations, $M = 100, M = 1000, M = 10000$. Compute the error in the solutions and compare them for the different values of M. What appears to be the convergence rate as a function of M?

4.2 A biochemical control network

Recently, stochastic models of biochemical reaction networks have attracted much interest in the field of (computational) *systems biology*, a field concerned with understanding cellular processes on a systems level. A cell, the domain where the reactions take place, is very small. In a popular model organism, the bacterium *E. coli*, the cellular volume is approximately $10^{-15}l$. Due to the small volume, some of the proteins involved in the reactions may be present in only $1 - 10$ copies. It has been illustrated in several papers that the classical ODE models based on concentrations or mean values fail to describe some properties of such systems as accurately as stochastic models can, and in this section we will study the model from one of those papers [6] in some detail. In particular, we will see how we can simulate biochemical networks stochastically.

The mathematical model we will be using is a *continuous time discrete space Markov Process (CTMC)*, and this is a very fundamental mathematical formalism with applications in numerous fields of applied mathematics, physics, chemistry and computer science. Even though we will formulate the algorithm to simulate such models in the context of biochemistry here, the same algorithm is a general method to simulate CTMCs, and thus has applications in other fields too, possibly under a different name.

The biochemical model we will study is a prototype model of a *circadian rhythm*. These kind of systems are responsible for adapting a cell or species to periodic oscillations in the environment, the most well known being the daily rhythm of approximately 24h. This model includes two genes, their corresponding mRNA and the proteins that are synthesized from the mRNA. However, before we can look at the properties of the model we need to develop the algorithm that we will rely on for simulations. We will also introduce some notation used when describing chemical reactions in a stochastic setting.

4.2.1 The stochastic simulation algorithm (SSA)

We write a chemical reaction between two chemical species A and B that react and form the species C as

$$A + B \xrightarrow{\omega(a,b)} C. \tag{12}$$

We assume *mass action kinetics*, and the *propensity function* $\omega(a, b)$ for the reaction is a polynomial, $\omega(a, b) = k_a ab$. This notation is common both in the deterministic ODE formalism and in stochastic models, but the models differ in the interpretation of a and b and $\omega(a, b)$. The first difference is that in the stochastic model a and b take values in the positive integers, $\mathbb{Z}_+$, and is

the *number of molecules* of the species, while in the deterministic model they are the *concentration* and thus real valued. Secondly, in the deterministic model, the propensity function would be interpreted as the rate with which the reaction occurs, and using the rates for all the reactions in the model we formulate equations for the rate of change of each species; the ODE system. In the stochastic model on the other hand, the propensity function is interpreted as the *probability per unit time* that the reaction occurs. Each reaction is then treated as a discrete jump from one state to another, changing the state by integer amounts. The reaction in (12) decreases A and B by one, and increases C by one.

Introducing the *stoichiometry vector* $\mathbf{n} = [-1, -1, 1]$ and the state vector $\mathbf{x} = [a, b, c]$, we write a reaction r as

$$\mathbf{x} \xrightarrow{\omega_r(\mathbf{x})} \mathbf{x} + \mathbf{n}_r. \tag{13}$$

A complete chemical system is then defined by stoichiometry vectors and propensity functions for all different reactions. Simple chemical systems modeled with the stochastic model can be studied with deterministic methods. The chemical master equation (CME) is a equation whose solution gives the probability function of the system. With this we can answer questions like, for example, what the probability that there are 10 molecules of A and 5 molecules of B, $P(A = 10, B = 5, t)$ at time t is. We can also compute not only expected values of all the species, but also all other moments such as the variance, from this function. In Figure 2 we show a solution of the CME with a deterministic method (like the ones studied in Scientific Computing III) for a simple two dimensional chemical system, i.e. the probability function for that system.

Whenever we can solve for the probability density with a deterministic method this is preferable; we obtain a lot of information of the system. However, as the number of different chemical species increases in the model, the harder it becomes to solve the CME. If we, for example, have 10 different proteins whose individual copy number can take values in $[0, 100]$, we have to solve a linear system with $100^{10} = 1 \times 10^{20}$ unknowns (!) using the deterministic method. When the models become this large, one then have to use a stochastic method. For more information about this aspect, see e.g. [5, 2, 4]

The way we can simulate the process *stochastically* is in principle very simple. First of all, a Markov process has the important property that, if we know the state at time t, the state at a time s later that t only depends on the state at t and not on any time before t. Sometimes this is called the *memoryless* property. The way we simulate the system is outlined in

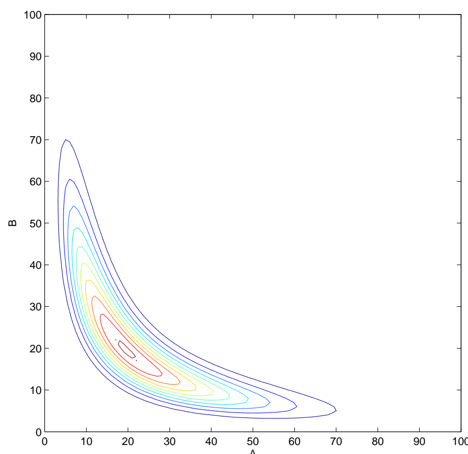


Figure 2: The probability density for a simple chemical system at time $t = 1000s$.

Algorithm 2. At time t, we generate the *inter event time* τ, the time until the next reaction occurs. We then sample which of the reactions $r = 1 \dots N_r$ that we should execute according to (13).

Algorithm 2 Simulation of the chemical system.

Initialize: Set the initial state $\mathbf{x} = \mathbf{x}_0$ and a final time T_f. Set $t = t_0$.
while $t < T_f$ **do**
  Sample τ, the time until the next reaction.
  Find the next reaction r that occurs.
  Update the state according to $\mathbf{x} = \mathbf{x} + \mathbf{n}_r$.
  $t = t + \tau$.
end while

What we need to know in order to be able to actually simulate the system is how exactly we can sample the random numbers τ and r in Algorithm 2. This was treated in a fundamental paper by Daniel T. Gillespie [3], and the algorithm we will come up with is often called the *Gillespie algorithm* or the *stochastic simulation algorithm* (SSA). We define $a_0(\mathbf{x})$ as

$$a_0(\mathbf{x}) = \sum_{r}^{N_r} \omega_r(\mathbf{x}), \tag{14}$$

i.e the sum of all propensity functions. The inter event time τ is *exponentially distributed* with mean $1/a_0, \tau \sim Exp(1/a_0)$. The PDF of the exponential

distribution is given in (15).

$$f(\tau; a_0) = \begin{cases} a_0 e^{-a_0 \tau}, & \tau \geq 0 \\ 0, & \tau < 0. \end{cases} \tag{15}$$

In the next exercise we will show how we can generate a random number τ from this distribution if we have access to random numbers from the uniform distribution $U(0, 1)$. This will tell us how to actually do step one in Algorithm 2. The method we will use is *inverse transform sampling* and is given in Algorithm 3.

Algorithm 3 Inverse transform sampling.

Generate a random number u from $U(0, 1)$.
Compute τ such that $F(\tau) = u$.
Take τ as a random number from the distribution defined by F.

$F(\tau)$ is the *cumulative distribution function* (CDF) and in the next exercise we will apply this procedure to generate random numbers from the exponential distribution.

Exercise 4

a.)

The CDF is defined as

$$F(t) = P(\tau \leq t) = \int_{-\infty}^{t} f(\tau; a_0) d\tau,$$

where $F(\tau \leq t)$ is the probability that τ is smaller that or equal to t. For the exponential distribution as defined in (15), compute the CDF.

b.)

Find the inverse of $F(\tau)$, i.e. for a number u find τ such that $F(\tau) = u$, or in other words $\tau = F^{-1}(u)$. *Remark: This procedure is general, but for many distributions there is no simple, closed form for the inverse CDF and the method may be computational expensive. For some important distributions there are other, more efficient methods available to generate random numbers.*

We are now ready to update Algorithm 2.
The only remaining issue before we have a complete algorithm to simulate the biochemical system is a method to generate the next reaction that occurs. The next reaction is a random variable Y with distribution [3]

Algorithm 4 Simulation of the chemical system.

Initialize: Set the initial state $\mathbf{x} = \mathbf{x}_0$ and a final time T_f. Set $t = t_0$.
while $t < T_f$ **do**
  Compute $a_0(\mathbf{x}) = \sum_{r=1}^{R} \omega_r(\mathbf{x})$.
  Generate a uniform random number u_1 from $U(0,1)$.
  $\tau = -ln(u_1)/a_0$.
  Find the next reaction r that occurs.
  Update the state according to $\mathbf{x} = \mathbf{x} + \mathbf{n}_r$.
  $t = t + \tau$.
end while

$$p(Y = r|X = \mathbf{x}) = \omega_r(\mathbf{x})/a_0(\mathbf{x}), \tag{16}$$

and the cumulative distribution function is given by

$$F(r; \mathbf{x}) = P(Y \leq r) = \frac{1}{a_0(\mathbf{x})} \sum_{k=1}^{r} \omega_k(\mathbf{x}). \tag{17}$$

Note that since $a_0(\mathbf{x})$ is the sum of all propensities $F(r; \mathbf{x})$ takes values in the interval $(0, 1]$. We can use Algorithm 3 to generate the random integer that gives the reaction in Algorithm 4. The difference between this case and the way we generated τ is that we cannot explicitly write down a formula to invert the CDF so we need to write a small algorithm that does this numerically. Given a uniform random number u_2, we generate a random reaction r by finding r such that

$$F(r-1) < u_2 \leq F(r) \tag{18}$$

or in more detail in our case, find r such that

$$\sum_{k=1}^{r-1} \omega_k(\mathbf{x}) < a_0(\mathbf{x}) u_2 \leq \sum_{k=1}^{r} \omega_k(\mathbf{x}). \tag{19}$$

Remark: The above procedure can be used to generate random numbers from other discrete distribution, e.g. the important Poisson distribution.

Computer exercise 9

Suppose that a `Matlab` function that computes the propensities $\omega_k, k = 1 \dots R$ is given

```
% PROPENSITY. Returns the value of the propensities
% for a simple 1D model.
%
% Input:  x - the current state of the system.
%
% Andreas Hellander, 2009.
%

function w = propensity(x)

k1=1e-3;
k2=1e-2;
mu=1e-1;

w=zeros(3,1);
w(1)=max(0,k1*x*(x-1));
w(2)=k2*x;
w(3)=mu;
```

Using this function, write a function that finds r according to (19), i.e.

```
% NEXT_REACTION. Finds the next reaction.
%
% Input:  re - A list with values of all propensity functions.
%         u  - A uniform random number in (0,1].
%
% Some Student, 2009.
%

function r = next_reaction(re,u)

    % Write this.
```

Hint: Read the help section for the functions 'cumsum' and 'find'.

At this point, we have all necessary components of the Gillespie algorithm, and it is stated in Algorithm 5. This version of the algorithm is the most commonly used formulation, and is called the direct method (DM).

Computer exercise 10

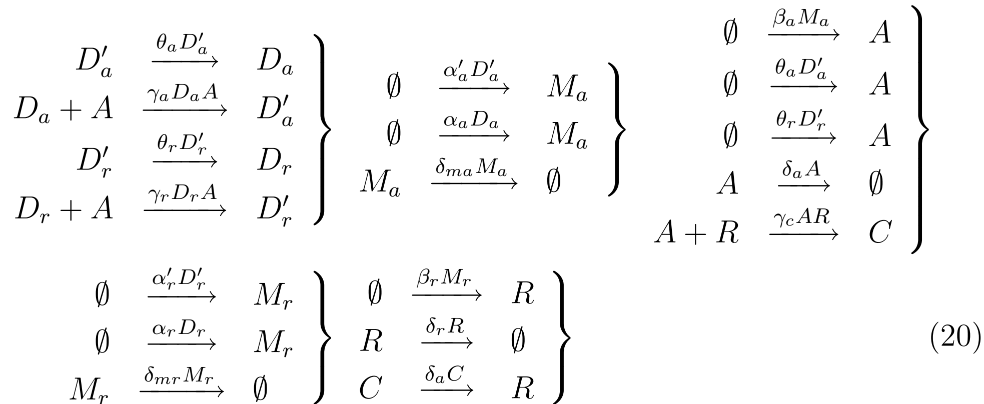
Algorithm 5 Gillespie's direct method (DM)

Initialize: Set the initial state $\mathbf{x} = \mathbf{x}_0$ and a final time T_f. Set $t = t_0$.
while $t < T_f$ **do**
  Compute $a_0(\mathbf{x}) = \sum_{r=1}^{R} \omega_r(\mathbf{x})$.
  Generate two uniform random number u_1, u_2 from $U(0, 1)$.
  $\tau = -ln(u_1)/a_0$.
  Find r such that $\sum_{k=1}^{r-1} \omega_k(\mathbf{x}) < a_0(\mathbf{x})u_2 \leq \sum_{k=1}^{r} \omega_k(\mathbf{x})$.
  Update the state according to $\mathbf{x} = \mathbf{x} + \mathbf{n}_r$.
  $t = t + \tau$.
end while

Implement Algorithm 5 and apply it on the propensities given by the function `propensity` in the previous exercise. The stoichiometry vectors are $n_1 = -2$, $n_2 = -1$, $n_2 = 1$. The input to the function should be a function that computes the propensities, a function that gives the stoichiometry matrix, the initial state and a vector that specifies the output times.

4.2.2 The circadian rhythm

Now that we have formulated and tested Gillespie's direct method on a simple model problem in Computer exercise 10 we will use it to study the more involved model in [6]. For the particular system we are considering here there are 18 reactions and they are found in (20).



$$\left.\begin{array}{rcl} D_a' & \xrightarrow{\theta_a D_a'} & D_a \\ D_a + A & \xrightarrow{\gamma_a D_a A} & D_a' \\ D_r' & \xrightarrow{\theta_r D_r'} & D_r \\ D_r + A & \xrightarrow{\gamma_r D_r A} & D_r' \end{array}\right\} \left.\begin{array}{rcl} \emptyset & \xrightarrow{\alpha_a' D_a'} & M_a \\ \emptyset & \xrightarrow{\alpha_a D_a} & M_a \\ M_a & \xrightarrow{\delta_{ma} M_a} & \emptyset \end{array}\right\} \left.\begin{array}{rcl} \emptyset & \xrightarrow{\beta_a M_a} & A \\ \emptyset & \xrightarrow{\theta_a D_a'} & A \\ \emptyset & \xrightarrow{\theta_r D_r'} & A \\ A & \xrightarrow{\delta_a A} & \emptyset \\ A + R & \xrightarrow{\gamma_c A R} & C \end{array}\right\}$$

$$\left.\begin{array}{rcl} \emptyset & \xrightarrow{\alpha_r' D_r'} & M_r \\ \emptyset & \xrightarrow{\alpha_r D_r} & M_r \\ M_r & \xrightarrow{\delta_{mr} M_r} & \emptyset \end{array}\right\} \left.\begin{array}{rcl} \emptyset & \xrightarrow{\beta_r M_r} & R \\ R & \xrightarrow{\delta_r R} & \emptyset \\ C & \xrightarrow{\delta_a C} & R \end{array}\right\} \tag{20}$$

The parameters of the model are given in Table 1. The complete propensity functions $\omega_r, \quad r = 1 \ldots 18$ are given above the arrows.

α_A	α'_a	α_r	α'_r	β_a	β_r	δ_{ma}	δ_{mr}
50.0	500.0	0.01	50	50.0	5.0	10.0	0.5
δ_a	δ_r	γ_a	γ_r	γ_c	Θ_a	Θ_r	
1.0	0.2	1.0	1.0	2.0	50.0	100.0	

Table 1: Parameters for the Vilar oscillator.

We have not explained this here, but from this description of the model, we can formulate a simple deterministic model for the expected values of the different chemical species. The *reaction rate equations* is a generally non-linear system of ordinary differential equations (ODEs)

$$\frac{dE[X]}{dt} = \sum_{r=1}^{N_r} \mathbf{n}_r \omega_r(\mathbf{x}). \tag{21}$$

For the model in (20), for example, the first of these equation would read

$$\frac{dE[D'_a]}{dt} = -\theta_a E[D_a] + \gamma_a E[D_a]E[A]. \tag{22}$$

The reaction rate equations are not exact equations for the expected values, but the approximation is very good when the participating species are present in high copy numbers. This is typically the case if the model comes from a chemical engineering problem where the volume is large (macroscopic) and the chemical species that are modeled are small molecules. Figure 3 shows the circadian system simulated with this deterministic model when the parameter $\delta_r = 0.2$ and when $\delta_r = 0.08$.

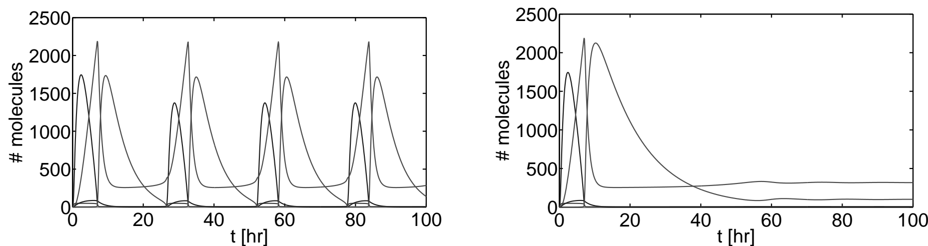


Figure 3: Deterministic simulation of the circadian rhythm model. When $\delta_r = 0.2$ (left) the clock oscillates with a period of $\approx 24h$, but when $\delta_r = 0.08$ (right) the clock stops.

As can be seen, when we lower the parameter (the degradation rate of the species R) to 0.08, the model stops oscillating. This would be a catastrophic

event for a living cell, and one would expect that the presence of oscillations would be more robust to changes in the parameters. Using this ODE model, however, we will inevitably get this sharp switch like behavior. In the paper [6], one also considers the stochastic model discussed in the previous section. We will now try to reproduce the conclusion from that paper.

Computer exercise 11

Simulate a single trajectory of the model defined by (20) and Table 1 using the algorithm you implemented in Computer exercise 10. Plot trajectories corresponding to $\delta_r = 0.2$, $\delta_r = 0.08$ and $\delta_r = 0.01$. Does the stochastic model display the same sharp "on–off" behavior as the deterministic ODE model does? Compare your simulations to the conclusions in paper [6].

5 Summary

In these lecture notes, we have seen examples where stochastic simulation and Monte Carlo methods were used to study different processes. We have also seen how to use pseudo random numbers to evaluate integrals. Monte Carlo becomes an alternative to deterministic methods for solving scientific problems in a number of different scenarios, but are often considered as a method of last resort; they are used when no deterministic methods is available or computationally feasible. The reason for this is the slow convergence of Monte Carlo methods ($\mathcal{O}(N^{-1/2})$), making them very computationally demanding if result with a high accuracy is desired.

For the integration example, it is not *in principle* hard to formulate deterministic quadrature rules to solve high dimensional integrals, but the number of quadrature points grows exponentially with the dimension. Their convergence rate is also sensitive to both the dimensionality and the smoothness of the integrand. When there is no longer possible to use these methods, Monte Carlo can still compute a result. However, no one would use Monte Carlo integration to compute an approximation to an integral in say two or three dimensions; deterministic methods are much more efficient in this case.

In another example of a biochemical control network we have seen that a simple stochastic *model* gives a better description of the system than the simple deterministic model. Even if we could formulate a complicated deterministic model that describe the system as good as the stochastic model, we rather use a simple stochastic model. Also in this case, depending on the question we are asking, we are presented with a choice between using a de-

terministic or stochastic *method* to study the stochastic model. Also in this case, if the deterministic method is applicable, we would use it. However, the computational work for the deterministic method grows exponentially with the number of different chemical species in the model, so for models with many species we have to resort to a stochastic simulation method, SSA.

Obviously, there are numerous of application areas in scientific computing where Monte Carlo can be used that we have not mentioned here, e.g. optimization, stochastic differential equations and solution of partial differential equations to name a few. However, it is the hope that you at this point understand the basic ideas behind Monte Carlo methods. The same ideas and workflow as we have seen here applies to many other situations, even though the details may be different.

References

[1] Steven C. Chapra. *Applied Numerical Methods with MATLAB for Engineers and Scientists.* McGraw–Hill, second edition, intenational edition, 2008.

[2] Stefan Engblom. *Numerical Solution Methods in Stochastic Chemical Kinetics.* PhD thesis, Uppsala University, 2008.

[3] Daniel T Gillespie. A general method for numerically simulating the stochastic time evolution of coupled chemical reacting systems. *J. Comput. Phys.*, 22:403–434, 1976.

[4] Andreas Hellander. Numerical simulation of well stirred biochemical reaction networks governed by the master equation. *Licentiate thesis, Department of Information Technology, Uppsala University*, 2008.

[5] Paul Sjöberg. *Numerical Methods for Stochastic Modeling of Genes and Proteins.* PhD thesis, Uppsala University, 2007.

[6] José M. G. Vilar, Hau Yuan Kueh, Naama Barkai, and Stanislav Leibler. Mechanisms of noise resistance in genetic oscillators. *Proc. Nat. Acad. Sci. USA*, 99(9):5988–5992, 2002.