Network Modeling and Simulation

Ilaria Cherchi
Telegram: @ilariacherchi
Giacomo Fantoni
Telegram: @GiacomoFantoni
Elisa Pettinà
Telegram: @elispettina
Github: https://github.com/giacThePhantom/network-modelling-and-simulation

January 4, 2023

Contents

Chapter 1

Introduction

1.1 General introduction

To cope with the inherent multi-physics and multi-scale natures of biochemical reactions, different levels of simulation detail have been adopted to investigate their dynamical behavior:

- **molecular dynamics** (MD): microscopic level - keeps track of the structures, positions, velocities, as well as possible collisions of all molecules in the system. The MD simulation requires a very detailed knowledge of molecules in both time and space and a lot of computational power.
- **Brownian dynamics** (BD): focuses on the dynamics of each individual species, but skips the molecular structure information. The movement of a species is described as a random walk (or Brownian walk) among point-like structures. The time scale of BD simulation is greatly improved over MD, but it is still limited when dealing with large models.
- **deterministic simulation**: highest coarse-grained approach which focuses on the macroscopic behavior of biochemical reactions. Molecular species in the deterministic simulation approach are represented by their concentrations. The rate of change in the concentration of each species due to a reaction is directly proportional to the concentrations of species involved in the reaction. The time evolution of a biochemical reaction network is described by a set of ordinary differential equations (ODEs). The deterministic simulation is fast; however, its underlying assumption inherently oversimplifies biological reactions in which populations of molecular species are continuous variables and their changes due to single reaction firings are assumed to be negligible. The correctness of deterministic simulation is severely affected when stochasticity plays an important role in the dynamical behavior of biochemical reactions.
- **stochastic simulation:** a mesoscopic approach to provide a probabilistic description of the time evolution of biochemical reactions. It keeps track of a discrete count for the population, but abstracts all the detailed position and velocity information of each species. Each reaction in the network is assigned a non-negative chance to fire and to drive the system to a new state. The probability that a reaction occurs in a time interval is derived from the reaction kinetics. Each stochastic simulation step will select a reaction to fire according to its probability.

Although the stochastic simulation is faster than the MD/BD approach, it is often computationally demanding for simulating large biological systems.

1. biochemical reactions, due to their multiscale nature, are separated by different time scales in which some fast reactions will occur at rates greater than other reactions. The dynamical behavior of biochemical reactions, after the short fluctuation time at the beginning, will be determined by the dynamics of the slow reactions; however, most of the time the simulation samples the fast reactions to realize the dynamics which is not the expected behavior.
2. the population of some species involved in reactions may be larger than others by many orders of magnitude i.e. the fluctuations of some species are more or less significant. Keeping track of large population species is obviously less efficient, since a coarse-grained simulation method can be applied without loss of total simulation accuracy. A model can combine and mix all of these aspects e.g. the fast reactions occur frequently and drive the system very fast into a stable state.
3. due to the stochastic behavior in a single simulation, many simulation runs must be performed to ensure a statistical accuracy and this requires a high computational effort. These issues raise a computational challenge for developing and implementing efficient stochastic simulation methods.

1.2 Network modeling

[missing lecture 2]

1.2.1 Logic models

Recap from the previous theory lecture: what is the main issue in using logic modeling with multiple levels? The update formulae need to be defined for each level, tricky extension procedure.

1.2.2 Petri nets

Petri nets are specific networks introduced in 1960s, with the idea to describe *communication processes* (computer science field). We have two kinds of nodes:

- ***places***: container of entities
  - *tokens*: entities
- ***transitions***: possibility to move one or more tokens to other places

In order to model a chemical reaction, we can associate places to variables, transitions to chemical transformations and tokens to molecules.

Figure 1.1: in the example we have numbers inserted in places. The network will evolve according to the transitions applied. A transition can be *enabled* or not to fire. E.g. there is one token in p_1, so we know that t_1 and t_2 are enabled. Instead, t_4 cannot be enabled, as 3 tokens are required but only one is present. t_2 is taking the token from p_1 and creating 2 tokens in p_2 and p_3; this allows to fire t_4, since now p_3 has the required number of tokens.

(a) shows the initial marking before firing the enable transition t;

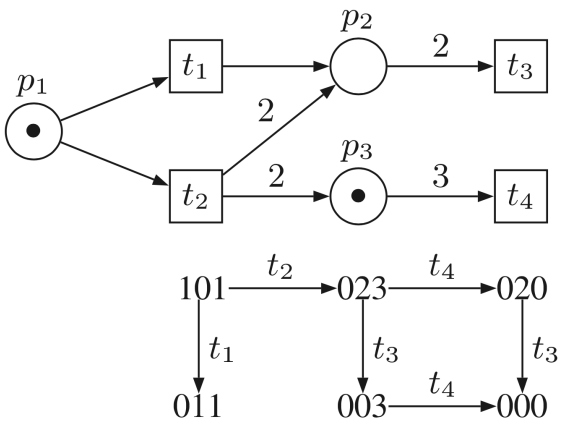

Figure 1.1: PetriNets_SimaoEtAl.pdf

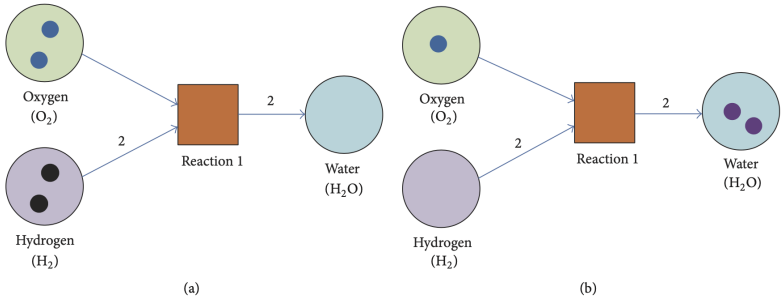

Figure 1.2: Review_ModelingComplexBiologicalSystems.pdf

(b) shows the marking after transition labeled reaction 1 fires. Places: hydrogen, oxygen and water. We can represent the stoichiometry of the reaction through the numbers on the edges and the numbers to the tokens.

Figure 1.2

- Pros: we have no constraint on the data type, not strictly boolean values. They allow to extend the number of items which can be associated to a model.
- Cons: there is no fixed rule for applying transitions. Furthermore, we are not encoding the reaction's complexity (all transitions are equally probable, but it is possible to weight the edges).

Having a dynamics based on the integers can be quite useful, as if we consider single chemical events we are working with discrete data. The exact stochastic algorithm works with integers. In differential equations instead we need real numbers. Also the discretization of the time step might not be a limitation, since time can be discretized in reality. The main limitation of network models is the **approximation of time**, as we are assuming that all the reactions take the same (unknown) amount of time.

By upgrading the notation we can achieve more accurate representations:

- add inhibitory and test transitions.
- differentiate between discrete and continuous transitions.

1.2.3 Rewriting systems

1.2.3.1 P systems

Popular rewriting systems are ***P systems***, which are also called membrane systems. They are computational environments inspired to the structure of membranes. In particular, they define a hierarchy of membranes partitioning the space in different areas - similarly to a cell. In each region we can allocate entities and apply transformation rules. The rewriting rules change the value of each letter. The pedix $_{in4}$ gives more details on the reaction. These kind of systems tend to use *non-determinism*, they try to explore the full set of possibilities.

1.2.3.2 MP systems

The difference with standard P system is the association of functions to each reaction. In this way we can reconstruct the complexity of the reaction, since in the model we apply all possible reaction - which will produce an amount given by the function.

1.2.4 Equation-based approach

1.2.4.1 ODE systems

Example: mass-action model $A + E \underset{k_2}{\overset{k_1}{\rightleftharpoons}} A \mid E \xrightarrow{k_3} B + E$

1.2.5 Simulation algorithms

For simulating we need the specification of a stoichiometric matrix, a vector of integers (initial values) and a stochastic rate. **Exact simulation algorithms** are computationally intensive, but provide the most accurate solution. Stepwise, we will try to define faster strategies with the aim of compromising accurate dynamics and feasible solutions. It is also possible to rely on a mixture of technologies to focus on different results. If the well-mixed assumption is not fulfilled:

- partition the compartment in sub-compartments → approximation
- use more sophisticated algorithms

The *stoichiometric matrix* tells us how the system evolves if one of the two functions is applied, but it is not enough for computing a simulation. There can be many reactions that arrive to the same definition of stoichiometric matrix. If we want to compute a dynamics we need to develop a series of states; at each step we require two ingredients:

- τ: tells how much later the system will evolve to another state
- μ: choose the reaction by considering the probability of execution of each reaction at the step

1.2.5.1 Reaction propensity

The reaction propensity is a function needed for the derivation of probability. The higher the propensity, the higher will be the strength of the reaction. Naturally, we will have a higher probability when a higher propensity is observed, but the two quantities are something different. The propensity is a property of the reaction, probability is a property of a reaction in the system (we need to take into account also other reactions) Instead of performing an in depth analysis of probability, we will choose a stochastic approach. It is only necessary to compute one evolution of the system per time.

Chapter 2

Stochastic Chemical Kinetics

2.1 Rewriting biochemical reactions

Biochemical reactions are the building blocks to model biological systems. They provide a unifying notation with sufficient level of details to represent complex biological processes. Biochemical reactions decorated with reaction kinetics can be simulated by a simulation algorithm to generate a realization of their dynamics. Chemical species in a biological system move around and gain kinetic energy. Upon collisions with other species, they undergo reactions to modify and transform into different species. In order to make this concrete, consider the transformation of a molecule of species A to a molecule of species B. It is written schematically as $A \rightarrow B$. This reaction converts one A molecule on the left side of the arrow to a B molecule on the right side. Such a transforming reaction is called a *unimolecular reaction*. We are dealing with a structure of compartments, in which we have symbols representing chemicals and *rewriting rules* specifying the direction. Membrane example: AABC A→B , A→C.

All these computational systems are more oriented on representation, while if we are more interested in the function we can apply **equation-based methods**, for instance ODE systems. In ODEs we specify the derivative in time of any of the variables in terms of a function of the state. The stochastic simulation approach can be approximated by a deterministic one.

The main steps to follow for writing a formal mathematical description of a biological system are**:**

1. Choose representation: arrow notation
2. State of the system: integer
3. Specify the likelihood of execution for each reaction

A reaction will be faster when we have a huge amount of reactants. Why do we state so? If we think in terms of probability, an example reaction A + B → C will be faster when a high quantity of A and B is present, since it is more likely for them to interact. Such statement is based on the assumption that there are no gradients in the system, i.e. well-mixed chemical system, therefore the distribution of the molecules is uniform.

2.1.0.1 Well-mixed reaction volume

Reaction volume in which all the molecular species are homogeneously distributed and spatially indistinguishable. Chemical species under the well-mixed assumption at a thermal equilibrium are uniformly distributed in the volume V and their velocities are thermally randomized according to the Maxwell-Boltzmann distribution. In order to approximate a complex system, we can partition the main compartment in smaller sets where we can apply this assumption - *discretization* procedure. In terms of stochastic representation we require integers e.g. probability that two molecules will interact. Therefore, Petri nets are a suitable representation for this kind of systems.

2.1.0.2 Formal representation

The state of a spatially homogeneous biological system is determined by the population of each species, while the position and velocity of each individual molecule are ignored. Let $X_i(t)$ be the population of species S_i at a particular time t. The N- vector $X(t) = (X_1(t), ..., X_N(t))$, which determines the population of each species, constitutes the system state at the time t. A general reaction R_j has a general scheme: $v_{j1}^- S_1 + ... + v_{jN}^- S_N \rightarrow v_{j1}^+ S_1 + ... + v_{jN}^- S_N$, in which a species on the left side of the arrow is called a *reactant*, while a species on the right side is called a *product*. The non-negative integers v_{ji}^- and v_{ji}^+ are the stoichiometric coefficients which denote the number of molecules of a reactant that are consumed and the number of molecules of a product that are produced. For each reaction R_j, the net change in the population of species S_i involved in the reaction is equal to $(v_{ji}^+ - v_{ji}^-)$, which can be positive, negative or zero. The net changes by all reactions are described by a stoichiometry matrix $\mathbf{v}$ with size MN. The jth row $\mathbf{v}_j$ of the stoichiometry matrix expresses the changes caused by reaction R_j and it is called the state change vector. We can have multiple systems leading to the same stoichiometric matrix.

2.1.0.3 Stoichiometric matrix

We require two matrices: V^+ provides the products, V^- provides the reactants.

$$V^- = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}, V^+ = \begin{bmatrix} 0 & 2 & 0 \\ 0 & 0 & 2 \\ 2 & 0 & 0 \end{bmatrix}, V = \begin{bmatrix} 1 & 1 & 0 \\ 0 & -1 & 1 \\ 1 & 0 & -1 \end{bmatrix} \; V = V^+ + V^-$$

Suppose that at a time t the state (number of molecules in that given moment) is $X(t)$. it further assumes that the next reaction scheduled to fire at the next time $t+\tau$ is R_μ, which moves the system accordingly to a new state $X(t+\tau)$. $\mathbf{x}$ is a simple notation to represent $X(t)$.

$$X(t+\tau) = X(t) + v_\mu$$

2.1.0.4 Summary

If we want to specify in computational terms the biological system that we are interested into, we end up with the structure of at least one compartment where we assume to have a chemical volume in which there are some entities that interact with each other. We impose a preliminary assumption, the well-mixed volume, that means that all these actors are present with equal availability in all the parts of the volume. For each compartment, we need to specify the entities (how many molecules are available), and the reactions that provide the rules for transforming the chemicals in others along the time. All these structures of reactions can be defined in mathematical terms using matrices: the

Definition 2.3: Mass action propensity

For mass action kinetics, the propensity a_j of reaction R_j in Eq. (2.1), given the current state $X(t)$ at time t, is

$$a_j(X(t)) = c_j h_j(X(t))$$

where c_j is the *stochastic reaction rate* and $h_j(X(t))$ counts the number of distinct combinations of reactants,

$$h_j(X(t)) = \prod_i \binom{X_i(t)}{\mathbf{v}_{ji}^-} = \prod_i \frac{X_i(t)!}{\mathbf{v}_{ji}^-!(X_i(t)-\mathbf{v}_{ji}^-)!}.$$

The number of combinations $h_j(X(t))$ of a synthesis reaction, where the stoichiometric coefficient of its reactants is zero, is set to $h_j(X(t)) = 1$.

Figure 2.1: Mass action propensity

number of columns is equal to the number of variables, the number of rows is equal to the number of reactions, the stoichiometric coefficient of reactant or products.

2.2 Reaction propensity

Each reaction in stochastic chemical kinetics is considered as a *stochastic process* where each of its occurrence is a random event with an assigned probability distribution. It is impossible to predict the progress of reactions deterministically, but only stochastically with a probability. The propensity of a reaction is a formula that is computed on a state of a system. It is the way that we will use to hint the probability of execution of the reaction.

The propensity a_j of a reaction R_j is defined such that $a_j(x)dt$ = probability that a reaction R_j fires in the next infinitesimal time interval $[t,t+dt)$, given the state $X(t) = \mathbf{x}$at time t. In a chemical setting, the probability of execution of one reaction will be proportional to the viability of the reactant.

2.2.1 Mass action propensity

The mass action propensity is the expression of the direct proportionality: multiply the rate of the reaction by the counts of the number of distinct combinations of reactants. The mass action propensity formula is reported in figure 2.1.

2.2.2 The propensity of a reaction

The propensity of a reaction is an intrinsic property of the reaction, is linked to the phenomenon that the reaction is going to represent. The probability will depend on the propensity of the reaction and the other reactions that are competing for the same reactant. The propensity is not affected by the products.

$$a_1(0) = c_1 h_1(x(0))$$

$$h_2(x(0)) = \binom{100}{1}\binom{50}{1}\binom{30}{0}$$

$$x(0+\tau) = x(0) + V_1 = \begin{bmatrix} 99 & 51 & 30 \end{bmatrix}$$

$$x(0) = \begin{bmatrix} 100 & 50 & 30 \end{bmatrix}$$

Here we are computing the combination, therefore we will not take into account a^2 as in the canonical law of mass action.

2.2.3 Reaction propensity for reactions R_j with mass action kinetics

- Synthesis reaction ($\emptyset \rightarrow$ products): the number of combinations $h_j = 1$ and propensity $a_j = c_j$
- Unimolecular reaction ($S_i \rightarrow$ products): the number of combinations $h_j = X_i$ and propensity $a_j = c_j X_i$.
- Bimolecular reaction ($S_i + S_k \rightarrow$ products): the number of combinations $h_j = X_i X_k$ and propensity $a_j = c_j X_i X_k$.
- Dimerization reaction ($2S_i \rightarrow$ products): the number of combinations $h_j = \frac{1}{2} X_i (X_i - 1)$ and propensity $a_j = \frac{1}{2} c_j X_i (X_i - 1)$.
- Polymerization reaction ($3S_i \rightarrow$ products): the number of combinations $h_j = 16 X_i (X_i - 1)(X_i - 2)$ and propensity $a_j = 16 c_j X_i (X_i - 1)(X_i - 2)$.
- Termolecular reaction ($2S_i + S_k \rightarrow$ products): the number of combinations $h_j = \frac{1}{2} X_i (X_i - 1) X_k$ and propensity $a_j = \frac{1}{2} c_j X_i (X_i - 1) X_k$.

2.3 Chemical Master Equation

The CME is the theoretical approach allowing to derive the complete set of probabilities of all the possible states of the system.

2.3.1 Grand probability function

$\mathbb{P}\{\mathbf{x}, t | \mathbf{x}_0, t_o\}$ = probability that the system state is $X(t) = \mathbf{x}$ at time t, given the initial state $X(t_0) = \mathbf{x}_0$ at time t_0. By applying the chemical master equation, we end up with a set of differential equations for this probability, which theoretically provide the complete set of any of the states of the system.

2.3.2 CME

$$\frac{\mathbb{P}\{\mathbf{x}, t | \mathbf{x}_0, t_o\}}{dt} = \sum_{j=1}^{M} (a_j(\mathbf{x} - \mathbf{v}_j) \mathbb{P}\{\mathbf{x}, t | \mathbf{x}_0, t_o\}) - \mathbb{P}\{\mathbf{x}, t | \mathbf{x}_0, t_o\} \sum_{j=1}^{M} a_j(\mathbf{x})$$

However, for computing all these probabilities in a complex system, the number of equations will increase and be practically useless. The idea is to avoid deriving everything with the chemical master equation but trying to apply another approach that is the one provided by the stochastic simulation.

Algorithm 1 Stochastic Simulation Algorithm (SSA) - General Sketch

Input: a biochemical reaction network of M reactions in which each reaction R_j, $j = 1,\ldots,M$, is accompanied with the state change vector $\mathbf{v}_j$ and the propensity a_j, the initial state $\mathbf{x}_0$ at time 0 and the simulation ending time T_{max}

Output: a trajectory of the biochemical reaction network which is a collection of states $X(t)$ for time $0 \leq t \leq T_{max}$.

1: initialize time $t = 0$ and state $X = \mathbf{x}_0$
2: **while** $(t < T_{max})$ **do**
3: set $a_0 = 0$
4: **for all** (reaction R_j) **do**
5: compute a_j
6: update $a_0 = a_0 + a_j$
7: **end for**
8: sample reaction R_μ and firing time τ from pdf $p(\tau, \mu|\mathbf{x}, t)$ in Eq. (2.16)
9: update state $X = X + \mathbf{v}_\mu$
10: set $t = t + \tau$
11: **end while**

Figure 2.2: SSA

2.4 Stochastic simulation

When you apply the CME, you explore all the possible state of the system. With a SS you compute only one possible trajectory of the system (for any possibility you must run several time a SS, too complex). The stochastic simulation works because you only need an idea of the most probable conditions of the system. SS is faster than a real experiment and you can run it with a computer (you can also run thousands of simulations).

2.4.1 Probability density function

The mathematical basis of a stochastic simulation is the reaction probability density function (pdf). $p(\tau, \mu|\mathbf{x}, t)d\tau$ = probability that a reaction R_μ fires in the next infinitesimal time interval $[t + \tau, t + \tau + d\tau)$, given the state $X(t) = \mathbf{x}$ at time t.

These probabilities can be computed by considering the idea of propensity. The propensity is a formula on the state of the system (depends on some of the variables that are in the system) that allows to provide a quantitative information that can be used to derive the probabilities. The propensity is not a probability: propensity is not in a range from 0 to 1 and it is a property of a reaction while the probability is a property of the system. One of the crucial points that we must focus on is being able to sample from this pdf given the fact that the computer has few ways of generating something that is stochastic. In particular, the easiest way to do so is through a random number generator. The pseudocode in figure 2.2 implements this procedure.
The result of a SSA run is a *trajectory*, which shows the evolution of the biological system over time. The trajectory is a collection of states $X(t)$ that denotes the state of the system at any time $0 \leq t \geq Tmax$. It should be emphasized that because SSA is a discrete event simulation algorithm, the state changes only at discrete time instants when reactions fire. The state between two reaction firings is a constant.

Chapter 3

Implementation of the Stochastic Simulation Algorithms

3.1 Non-deterministic vs stochastic

Assumption: we are using the same model and parameters.

- **Deterministic system**: no randomness, we always obtain the same result.
- **Non-deterministic system**: there is some degree of uncertainty on different runs.
  - **Exact stochastic simulation**: satisfying some hypotheses, the system will behave just like a biological system. We are in a probabilistic setting. The fact that we are able to compute the probability function does not make this method deterministic, we have uncertainty. Theoretically, we could have no idea about reaction execution in a stochastic setting; in the case of exact stochastic simulation, we reach a high level of accuracy thanks to the probability function.

Why do many computational environments employ a non-deterministic approach? Quite often we require to find the compromise between time and complexity. In non-deterministic polynomial time algorithms we do not have an efficient solution, but it seems possible to find it. A non-deterministic setting allows us to understand whether an algorithm can be solved in polynomial time (stepwise guessing). Alan Turing realized that it was required to categorize algorithms with specific rules in order to compare them → Turing machine.

A summary of the main exact stochastic simulation algorithms is reported in figure 3.1.

3.2 Direct method

Gillespie's direct method defines a couple of formulae able to understand how the system will execute in terms of time (τ) and reactions (μ). Since we are reasoning in infinitesimal time, each reaction occurs and ends exactly at time τ, hence we cannot have multiple reactions firing simultaneously.

a_0 is the sum of all propensities in the system.

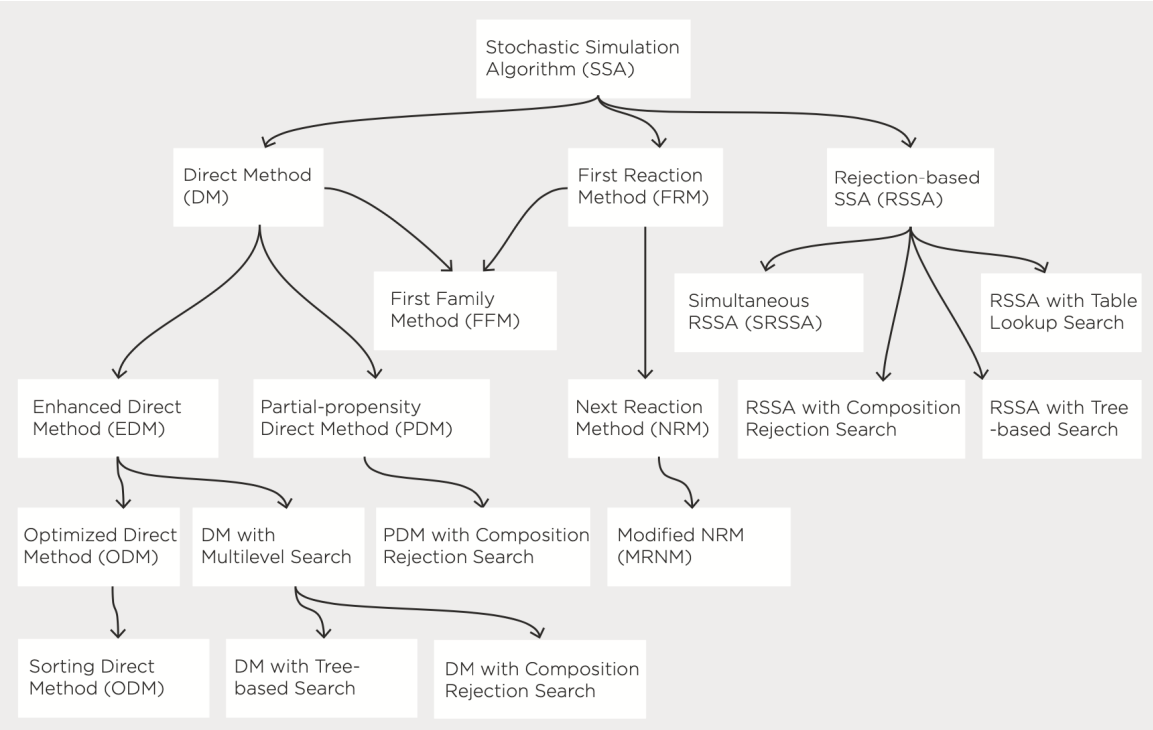

Figure 3.1: Fig 3.1 Marchetti's book

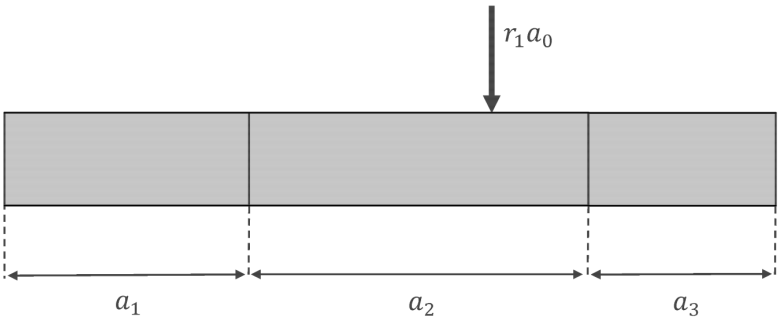

Figure 3.2: Fig 3.2

Algorithm 2 Direct Method (DM)

Input: a biochemical reaction network of M reactions in which each reaction R_j, $j = 1, \ldots, M$, is accompanied with the state change vector $\mathbf{v}_j$ and the propensity a_j, the initial state $\mathbf{x}_0$ at time 0 and the simulation ending time T_{max}

Output: a trajectory $X(t)$, $0 \le t \le T_{max}$, of the biochemical reaction network

initialize time $t = 0$ and state $X = \mathbf{x}_0$
while $(t < T_{max})$ **do**
  set $a_0 = 0$
  for all (reaction R_j) **do**
    compute a_j
    update $a_0 = a_0 + a_j$
  end for
  generate two random numbers $r_1, r_2 \sim \mathrm{U}(0,1)$ (see Appendix B.1)
  select R_μ with the smallest index μ such that $\sum_{j=1}^{\mu} a_j \ge r_1 a_0$
  compute $\tau = 1/a_0 \ln(1/r_2)$
  update state $X = X + \mathbf{v}_\mu$
  set $t = t + \tau$
end while

Figure 3.3: DM algorithm

1. We sample one random number from the distribution $a_0 = \sum_{j=1}^{M} a_j \rightarrow V_1 = U(0,1)$
2. Scale it to the maximum $V_1 \cdot a_0 = U(0, a_0)$
3. See where this number will point over the different propensities (Figure 3.2).
4. Generate another random number $V_2 = U(0,1)$
5. $\tau \sim Exp(a_0)$
6. $\tau = \frac{1}{a_0} ln(\frac{1}{V_2})$

The algorithm (Figure 3.3) can be improved by avoiding to recompute all the propensities, taking into account the fact that many reactions will keep the same probabilities over time. We can work on the complexity of the for loop through a *dependency graph*.

3.3 Enhanced Direct Method

Our aim is to group all the reactions that are modifying at the same time, in order to update only meaningful propensities e.g. all reactions involving the same components. Let's specify this concept

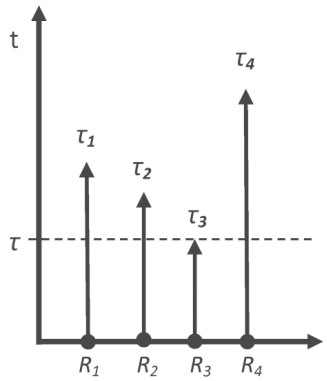

Figure 3.4: R Tau

in a formal way:

For each reaction $*R_j*$ with $j = 1, ..., M$, define $Reactants(R_j) = S_i|S_i$ is a reactant of R_j, and $Products(R_j) = S_i|S_i$ is a product of R_j. A **reaction dependency graph**....

We tend to have an improvement in time complexity when the graph is not highly connected, while the space complexity is higher (since we need to generate a data structure).

3.3.1 Improvements for the direct method

Sort: find the smallest index for which we can satisfy the condition for μ. On average, it is more efficient to keep reactions with highest probability at the beginning, such that we can easily match indexes to requirement. How can we compute the number of firings? We can set an oder beforehand based on previous knowledge or run a pre-simulation with a non-optimized algorithm (we still have a speed up since we are working with a smaller setting). We cannot directly compute the propensity right away, as it heavily depends on the dynamics of the system, the state can change. It is required to compute some heuristics to sort the reaction order → standard Gillespie. It is also possible to apply a ***multi-level search***. We group the reactions and select again a reaction in the group. As the dependency graph, here the issue is time complexity.

3.4 First Reaction Method (FRM)

Instead of computing one τ, compute a τ for each reaction (Figure 3.4). Example: $\tau_1 = Exp(a_1) = 1/a_1 ln(1/V_1)$. We are assuming that no other reactions are firing in the middle. We can generate M random numbers and end up with M τ. Then we choose the reaction with minimum τ, which will be the first selected one. $\mu = R_\mu$ st $\tau_\mu = \min_j \tau_j$

- Pro: sometimes the search is quicker, simpler to parallelize.
- Cons: we generate a lot of random numbers with respect to the direct method.

After the first step, we need to recompute. Even here, there is the possibility to improve the algorithm e.g.computation of the propensity.

3.5 First Family Method

Tries to combine the good features of the direct method with the first reaction method. The idea is to reach an implementation which can be partially parallelizable: divide the reaction in "families" or groups. The idea is to divide the reaction in n groups $r_1, \dots r_n$ and families e.g. 3 families. We should associate a theoretical propensity to each group, i.e. the sum of the propensity in the family $a^1 = \sum_{j \in f_1} a_j$. $FRM = \tau_1 = \frac{1}{e^1} ln(\frac{1}{r_1})$, we do not know which one of the reactions will be applied.

In order to decide, we apply the direct reaction method formula for selecting the index. Given that we are working with a subset, we will scale over the sum, the a_0 of the group. It is true that we are generating random numbers, but over the number of the families → reduction. This time we have n families + 1 random number to select the family. Danger: if we have big disequilibrium in propensity, we might end up selecting always one of the families. We can parallelize by linking families to CPU.

Last time, we were introducing Next Reaction Method, which can be considered as an evolution, trying to apply the same reasoning as FRM in a more efficient way (by applying an efficient handling of random numbers). If we are able to do so we will need to compute less random numbers. DM → $2 \cdot n_{steps}$ FRM → $M \cdot n_{steps}$ FFM → $(n_{families} + 1) n_{steps}$ NRM → $M + n_{steps}$. If we take a look at the difference among methods, it is not very clear which is the winner - even though NRM seems to be one of the most optimized, it is usually the most efficient in requiring less random numbers. Of course everything has a price, we need to add computations. NRM is the most efficient in random number generation, while FRM is the one consuming the most.

3.6 Next Reaction Method

Why do we need to recompute time points? Two issues:

1. multiple reactions can be executed at the same time → the reaction propensities might change
2. we are computing τ over propensities, but the propensities were computed on initial reaction settings. E.g.$R_2 : A \to B$, $R_1 : A \to B$, the two reactions depend on each other.

τ is modelling the instant in which the reaction is assumed to fire, it is just an event. We can subtract the time for passing to following reactions, but we must also update the propensity. $a_1 \to a_1^{new}$, we have three possibilities:

1. $a_1 = a_1^{new}$. R_2 is not affecting the propensity, i.e. reactant is not modified by R_2, so R_1 remains unchanged.
2. $a_2^{new} > a_1$. R_2 has been applied, something has changed and the firing prob of R_1 is increasing. Therefore, τ_1 needs to be updated, we expect it to be smaller → rescaling through the ratio of the propensities e.g. $(\tau_1 - \tau_2) \cdot \frac{a_1}{a_1^{new}}$
3. $a_1^{new} < a_2$ higher τ_1

We should not think of another event, we are just updating the same event through the formula! The algorithm was developed in the 2000s, there are a lot of improvements related to technology advancements. In addition to rescaling formula, there are other improvements to keep the complexity of the algorithm as low as possible:

- we avoid generating too many random numbers
- reaction dependency graph (introduced for Direct Method): recompute the propensity of the reaction only if it is needed
- searching the reaction: minimum is linear over the number of reactions. **Binary heap**: decrease to logarithmic scale, select winner in constant time.

Gibson 2000: original paper Thanh 2014: latest algorithm.

3.7 RSSA

The Rejection-based Simulation Algorithm seems to be an approximation, but everything is exact. It tries to reduce computation complexity, but instead of limiting the amount of random numbers, it focuses on another bottleneck: the computation of the reaction propensity.

The computational problem becomes more complex if we do not rely on mass action. The RSSA Algorithm therefore tries to reduce the number of times in which we compute the reaction propensity. We will need a placeholder and at the same time to keep the computation exact. Instead of using the state of the system, we use a ***fluctuation interval***(a bound). From the state of the system X we generate a lower and upper bound; to do so, we just use a parameter providing a percentage, which is called δ $(0 < \delta < 1)$. Therefore, the bounds will be computed as:

- lower $X = X - X \cdot \delta$
- upper $X = X + X \cdot \delta$

Given that the events of the reaction are modifying a few molecules per time, we can assume that for a certain amount of steps we will remain inside the same range of variability. We are trying to obtain a propensity computed over bounds; by doing so, we can apply the same computation for a higher number of steps, as we only need to recompute the propensity outside of the fluctuation interval. We can also compute the upper bound for the sum of the propensities $\bar{a}_0$. a_1 will be in the middle, but we do not want to compute it. It seems like working with these bounds does not lead to an exact simulation: we need to avoid as much as possible the exact computation of the propensities.

Indeed, in the vast majority of cases we will have the possibility to choose the reaction without the real propensity. We will perform some computations and, given the presence of bounds, we will do some mistakes; the algorithm will evaluate the results and eventually reject them (rejection-based algorithm). For doing so, the algorithm generates the events with the upper bound of the propensity $\bar{a}_0$. By using an upper bound τ will be lower, so we will generate more events with respect to the direct method. We apply a sort of Gillespie, we replace the standard propensity with the upper bound and choose the reaction as in the direct method. How can we understand whether an event is real or fake? In the case in which $\bar{a}_0 = a_0$ we are in the standard Gillespie. We should apply a similar reasoning as NRM for scaling for checking the validity of candidate reactions.

Let's imagine that we are selecting R_2: we should generate another random number, scale it over the bound and understand if we are in the exceeding (fake) or inside (real) area.

- If the random number is lower than the lower bound, we can accept it without computing the real propensity.

- If the random number is between the bounds we require the exact propensity for discriminating:
  - $\underline{a}_2 < u < a_2$ accept
  - $a_2 < u < \bar{a}_2$ reject

The higher the fluctuation interval, the higher the uncertainty but lower the complexity $\rightarrow$ find the right compromise with some heuristics on δ value.

Chapter 4

Approximation algorithms

Can we assume that in any condition the stochastic simulation bottleneck is a problem? It depends. If we are in a condition in which all species are present in low number and it takes a lot to cure, the stochastic simulation is not really a bottleneck. We might have an issue if we wish to simulate for a long time, as we require multiplicative factors. At the moment, it is not feasible to compute an exact stochastic simulations for all the models that we need to observe, in particular for a huge number of species and reactions. Therefore, there is a huge research in developing alternative strategies that allow to compute the system in an *accurate* way (lose correctness) with lower computational cost and time. Why should we decide to use approximations in simulations?

1. it might be the only possible and feasible solution to solve a problem.
2. reality is affected by error, so even when we are using exact stochastic simulation algorithms we should consider a small degree of approximation. A certain deal of error could aid in retrieving a more realistic result.

In chapter 3 all the algorithms were belonging to exact stochastic simulation setting, so the result is the same. Instead in this case, each algorithm (Figure 4.1) approximates in a different way, we assume different approximations $\rightarrow$ not a consistent set. We should observe that of the three main computational strategies presented here, there is one which is much more popular with respect to the others. The τ leaping method is well known and has been further developed, so we will be focusing on it.
Before introducing the algorithm, let's try to think about the possible bottlenecks which could prevent us from having a fast algorithm. We could improve:

- work on the number of reaction events, group them in such a way to reduce the number of reactions in the system. The τ leaping methodology is working in this direction
- improve the computation of the propensity. For instance, the probability-weighted Monte Carlo is particularly promising in situations in which we have huge differences in propensities among reactions.

4.1 Probability-Weighted Dynamic Monte Carlo Method

The probability-weighted dynamic Monte Carlo method (PW-DMC) is an approximation approach for improving the computational efficiency of stochastic simulations of reaction networks where some

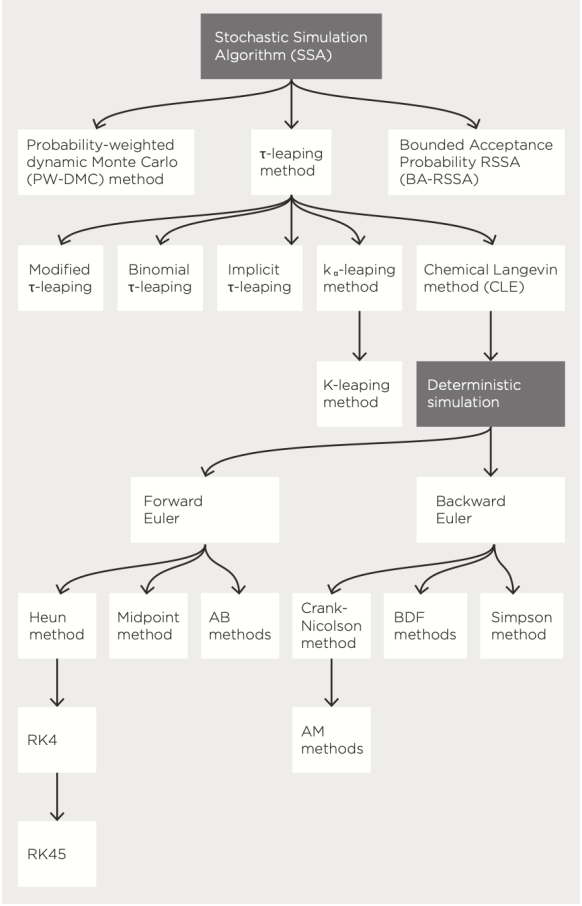
Stochastic Simulation Algorithm (SSA)
Probability-weighted dynamic Monte Carlo (PW-DMC) method
τ-leaping method
Bounded Acceptance Probability RSSA (BA-RSSA)
Modified τ-leaping
Binomial τ-leaping
Implicit τ-leaping
k_θ-leaping method
Chemical Langevin method (CLE)
K-leaping method
Deterministic simulation
Forward Euler
Backward Euler
Heun method
Midpoint method
AB methods
Crank-Nicolson method
BDF methods
Simpson method
RK4
AM methods
RK45

Figure 4.1

reactions have propensities significantly larger than other reactions. The principle of this algorithm is a sort of modification of the probability distribution NRM through *weighted sampling*. The idea is to introduce a bias weight value on the propensity, in order to take into the account the number of times in which the reaction is executed.

We will have a lower amount of iterations at the price of applying reactions a different number of times according to the probability. The weight is a multiplicative factor of the stoichiometric coefficient. μ is the smallest reaction index μ such that:

$$\sum_{j=1}^{\mu} a_j^w \geq r_1 a_0^w$$

Second, PW-DMC has to correct the firing time τ of reaction R_μ because it is selected through the weighted sampling. The generation of τ is adapted from SSA in which it is generated from an exponential distribution with rate a_0^w. PW-DMC then updates the state at time $t + \tau$ by assuming that there are w_μ consecutive firings of R_μ in the time interval $[t, t + \tau)$. This methodology started to provide some hints for the approximation setting. In the complete algorithm we have the concept of effective propensity, allowing to reach the speed up.

4.2 Bounded Acceptance Probability RSSA

If we compare RSSA and DM algorithm, the main difference is that the RSSA algorithm is more complex than DM at each iteration. The bottleneck in RSSA is that we use the third random numbers for the rejection test, so this algorithm tries to speed up the process and reduce the complexity. If we decide to accept everything, we will end up with a strong bias. In order to avoid the high computational costs, we can only apply rejection to some cases $\rightarrow$ set up bounds, reaching bounded acceptance probability RSSA. If the probability of the candidate reaction is greater than certain amount, we accept it without the rejection test. Algorithm: we are still executing one reaction at a time. Comparing in terms of accuracy, the approximation of the previous method is not applied. Here we are not working with the real propensity. In this version we are totally avoiding the rejection-based part, the approximation in this case increases the error. It could be possible to apply a strategy in which we apply rejection based only in certain conditions (more complex). By looking at the literature, it is interesting to rank strategies in terms of accuracy and time.

4.3 τ-Leaping Method

The τ-leaping method is a stochastic approximate algorithm for improving performance of stochastic simulation. Its aim is to discretize the time axis into time intervals and to approximate the number of reactions firing in each time interval. The simulation then leaps from one time interval to the next interval with many reaction firings performed simultaneously. What is remarkable here, is that we are doing an approximation over the number of iterations which is much more relevant than previous approaches. We are grouping a mixture of events, which can be defined as a *macrostep*. If τ is big, the number of iterations will be low and the complexity will decrease. We should constraint the error in such a way that we are aware of its value as a constant. Which could be a high-level property of tau to apply macrostep without a crude error in the simulation? We should impose certain logics on τ.

We want to limit the bias on the propensity. This translates in tau selection satisfying the so-called "**leap condition**":

There exists a leap $\tau > 0$ such that the change in propensity a_j of each reaction R_j with $j = 1, ..., M$ during the time interval $[t, t+\tau)$ is negligibly small. Negligibly small means lower than some threshold.

By applying a proper reasoning we reach the following formula, in which we derive the update of the state. Once we have selected τ, we will evolve the system by adding to he current state a certain amount of firing for each one of the reactions in the system [for each reaction j from 1 to m we apply a certain amount of time].

$$X(t+\tau) = \mathbf{x} + \sum_{j=1}^{M} k_j \mathbf{v}_j = \mathbf{x} + \sum_{j=1}^{M} Poi(a_j(x)\tau)\mathbf{v}_j$$

k_j will be generated through a Poisson distribution, which will depend over the specific propensity multiplied by the size of the step. The algorithm is applying any of the reactions per a specific amount of time.

- Pros: we are allowed to work with higher τ with respect to exact stochastic simulations.
- Cons: if we only have rare events, the algorithm will not be the best solution (as it will try to apply all reactions at all the times).

Given however that in any system there are at least some reactions with high propensity, at the very end we still gain a huge speed-up.

4.3.1 Choosing tau - Leap selection section

4.3.1.1 Postleap Selection

Choose a τ from a Poisson distribution and see if it is fine for the simulation. We start from a predefined arbitrary τ (small), then we have a trial procedure. The algorithm will compare the difference in the propensity before and after, to assess whether τ is of the right size. E.g. if the change in propensity is small, we can use a greater τ. The complexity of the τ-leaping is moved to a sub-problem, which is the choice of τ . We could also perform a preleap selection or choose other strategies. The Poisson distribution could provide more events than the ones that are possible in the system; in this case we end up with a negative population. While performing τ-leaping, we will need to make sure that a negative scenario is never reached through a test. The real amount of time executing k should be computed as the mean from k generated from Poisson distribution and the availability of the reactants [non chiaro]. We can bridge τ-leaping with exact stochastic simulation when we have only rare events.

4.4 Chemical Langevin Method

We can use the Normal distribution instead of Poisson, and scale it with mean and variance with another Normal formula based on a (0,1) distribution, obtaining a simplified method. We will not have main events, but main *drivers* in the system. Stochasticity is strongly approximated, we are moving to an idea of average, real numbers. This will be the starting point for the deterministic

simulation.

COPASI is a well-known simulator. It exploits Gillespie's algorithm and τ-leaping algorithm. In general, any simulator offers many algorithms, so we should be able to choose the best computational approach $\rightarrow$ calibrating the model. The stochastic rate provides us the speed, we require to derive it with some methods; all simulations algorithm compute deterministically or stochastically a speed for the reaction, computed over parameters of the model e.g. the rates. The rates, as the initial values, are not free-lunch, they can be derived through experiments. Often times, we work with models with not fully known parameters; in the second module we will explore solutions for *estimating the parameters* (reverse engineering approach).

Adaptive algorithms select a delta and change it adaptively during the simulation in order to reach the best results, for instance post leap τ selection. **Chemical Langevin Method** allows to modify the evolution of the system; the update formula depends on the propensity, but allows to distinguish a totally deterministic part from the stochastic one. This algorithm also has an important difference with previously seen strategies: we do not have integers, we will be provided with a real number adding some noise. This dynamics is over continuous numbers, tends to an average behaviour.

Chapter 5

Deterministic simulations

A *deterministic* way of calculating the dynamics of a system, when the last (noise) term of CLM equation becomes negligibly small compared with the second one, can be applied. This happens in the limiting case $a_j(\mathbf{x})\tau ß\infty$, $j = 1, ..., M$, and the deterministic simulation produces an average behaviour of the system that is very close to the one that results by averaging an infinite number of stochastic simulations of the system starting from the same initial state. If we work in terms of *moles*, we are moving Avogrado numbers; it is not so distant from the reaction size, the assumption is correct enough.

The deterministic simulations should give as an output the same result as an average stochastic simulation. ODEs can be safely used to simulate a biochemical system that satisfies the spatial homogeneity and continuum hypothesis.

5.0.1 Continuum hypothesis

"A biochemical system satisfies the continuum hypothesis if the number of molecules for each species is large enough to safely approximate molecular abundances by concentrations that vary continuously (as opposed to integer- valued molecule counts)." Sometimes, models simply use the ODE formalism, regardless of the hypothesis (since the ODE is the only computational framework allowing to reach the end of the simulation).

We are required to apply conversions and derive the ODE according to the following tables.
In Figure 5.1 here we are not considering the combination of the reactants. c = stochastic rate, k = deterministic rate.

Reaction type	Reaction	Rate	ODEs
Zero-order reaction	$\emptyset \xrightarrow{k} A$	k	$\frac{d[A]}{dt} = k$
First-order reaction	$A \xrightarrow{k} B$	$k[A]$	$\frac{d[A]}{dt} = -k[A]$; $\frac{d[B]}{dt} = k[A]$
Second-order reaction	$A + B \xrightarrow{k} C$	$k[A][B]$	$\frac{d[A]}{dt} = \frac{d[B]}{dt} = -k[A][B]$; $\frac{d[C]}{dt} = k[A][B]$
Second-order reaction (same reactant)	$A + A \xrightarrow{k} B$	$k[A]^2$	$\frac{d[A]}{dt} = -2k[A]^2$; $\frac{d[B]}{dt} = k[A]^2$
Third-order reaction	$A + B + C \xrightarrow{k} D$	$k[A][B][C]$	$\frac{d[A]}{dt} = \frac{d[B]}{dt} = \frac{d[C]}{dt} = -k[A][B][C]$; $\frac{d[D]}{dt} = k[A][B][C]$

Figure 5.1: Table 4.4 Marchetti's book

Reaction order	Reaction	Deterministic rate constant	Unit
Zero-order reaction	$\emptyset \xrightarrow{c} A$	$k = c/(N_A V)$	$concentration \cdot time^{-1}$
First-order reaction	$A \xrightarrow{c} B$	$k = c$	$time^{-1}$
Second-order reaction	$A + B \xrightarrow{c} C$	$k = cN_A V$	$concentration^{-1} \cdot time^{-1}$
Second-order reaction (same reactant)	$A + A \xrightarrow{c} B$	$k = cN_A V/2$	$concentration^{-1} \cdot time^{-1}$
Third-order reaction	$A + B + C \xrightarrow{c} D$	$k = c(N_A V)^2$	$concentration^{-2} \cdot time^{-1}$

Figure 5.2: Table 4.5 Marchetti's book

As shown in Figure 5.2 it is possible to go back to the stochastic setting from the deterministic one. CLM equation: if we assume that the product is crazily high i.e. close to infinite, we can assume that the two parts of the equations have different orders; in this specific condition, the noise part becomes negligible.

5.1 Deterministic approximation

If the propensity is approaching infinite, we can assume that the deterministic setting is motivated. We are required to introduce the law of mass action and employ tables [last lecture] to derive a set of ODEs. In this context, the law of mass action is a bit different from what we have observed at the beginning of the course. We need to remember that the constant of proportionality, i.e. rate, is not the same as the stochastic setting.

$$\frac{d[S_i]}{dt} = \sum_{j=1}^{M} (k_j \mathbf{v}_{ji} \prod_{l=1}^{N} [S_l^{\mathbf{v}_{jl}^-}]), i = 1, ..., N.$$

The equation $\frac{d[S_i]}{dt}$ should represent a molar concentration over time. We have an equation for each of the species $i = 1, \ldots, N$. It should take into consideration the effect of all substances in the system $\rightarrow$ sum over M, considering all reactions, which will be multiplied by the stoichiometric index. The system is computed as k (constant of proportionality), stoichiometric index and the product of all the reactants. If a species is not a reactant, $S_i = 0$, we end up with 1, only product of the reactants.

$$\frac{d[A]}{dt} = -\frac{d[B]}{dt} = -V_{MAX} \cdot \frac{[A]}{K_M + [A]}$$

We can just rely on a simplified set of equations to avid mass action correspondence. V_{MAX} = maximum velocity of the enzymatic reactions, K_m, called *Michaelis constant*, indicates the concentration of the substrate at which the reaction rate is half of its maximum value.

5.2 Numerical solution of ODEs

Often times in computational biology it is unlikely to have enough power for deriving an analytical solution of the system, reality is different and the complexity prevents this kind of approach. Therefore we rely on numerical methods to compute the simulation, adding an additional layer of approximation. The algorithm uses the idea of derivative to understand the next value. For a value of $t > 0$, the level of approximation could be remarkable and depends on the dynamics of the system. Remember that when we simulate in time we have the issue that the computation of the next state

depends on the previous state: we are dealing with an iterative formula, so potentially we can have a huge error explosion. At the basis of such method is the concept of derivative and geometrical rule to identify next step; Euler's method is the simplest one.

5.2.1 Euler's method

The first examples of explicit/implicit numerical methods are the *forward Euler method/backward Euler method* for updating the system state:

- Forward Euler: $[X_{n+1}] = [X_n] + hF(t_n, [X_n])$
- Backward Euler: $[X_{n+1}] = [X_n] + hF(t_{n+1}, [X_{n+1}])$

Not surprisingly, for increasing the order of the algorithm we will require a more accurate approx of the derivative, translating in additional evaluation of the ODE.

5.2.2 RUNGE-KUTTA Method

RK4 is a 4th order method, good compromise in accuracy and computational requirements. This algorithm uses a sort of average over Ks, which are computed by solving the set of ODEs as following:

$K1 = F(t_n, [X_n])K2 = F(t_n{+}2h, [X_n]{+}h2K_1)K3 = F(t_n{+}h2, [X_n]{+}h2K_2)K4 = F(t_{n+1}, [X_n]{+}hK_3)$

. Is it possible to increase accuracy without increasing complexity?

5.2.3 Midpoint method

2nd order method, we compute the new state at line 5 by two elements: x the current state and x old (the previous one). In this case, the multi-step method links the increased order to the number of step previously considered to compute the slope.

Drawback: what happens in the first step? We require an initial state and and additional time series! Limitation of multi-step algorithms: they all require a discretization step, it rules the movement. The time should not be computed during the iteration, chosen from the beginning; if we think about this, it can be a limitation of the application of the methodologies, which ask the user an information which might be unknown.

Which is the right time for simulating? Of course we can try to keep the discretization step as small as possible, but the right question that should be asked is: which is the error value? Keeping the error below a certain threshold is not easy, since the dynamics evolve with the system. In order to achieve error control, we should apply more requirements to discretization, leading us to seek an *adaptive solution*. In order to derive a rough estimate of the error we should compare the state computed by two algorithms of different order: one should be more accurate than the other, to have an idea of the degree of approximation.

5.2.4 Adaptive methods - Runge-Kutta-Fehlberg

"Strapopular algorithm: Runge-Kutta-Fehlberg, RK45 for friends". RK45 is composed of two methods, 4th and 5th order to apply error evaluation [4th for dynamics, 5th for estimating the error]. The algorithm will scale the step (h), in such a way that we keep the deviation below an error threshold.

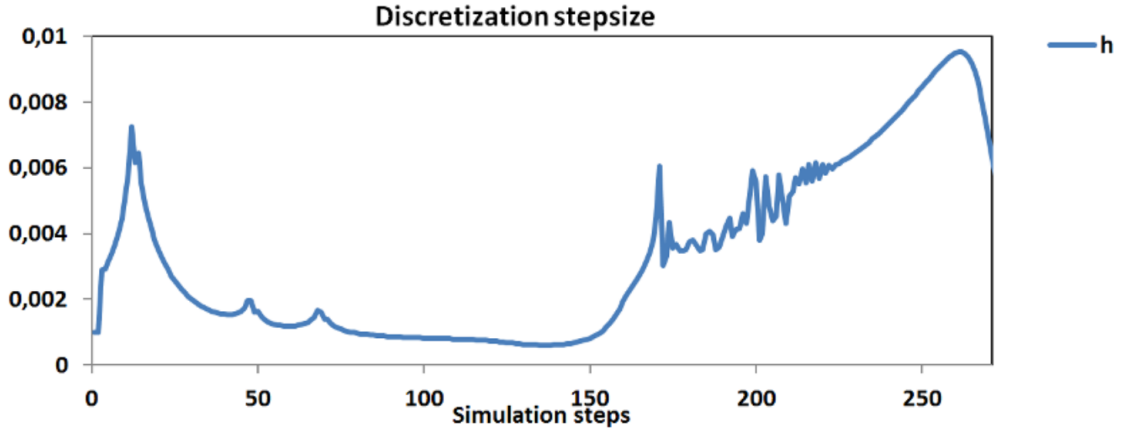

Figure 5.3: Figure 4.9 Marchetti's book

A formula will allow to play with h according to the deviation (increase if deviation is little). Recall that for the 5th order we require 6 derivations.

The two methods are similar, they share the same Ks, therefore at the price of computing a fifth order method we will have the possibility to obtain two evaluations → quite efficient method. ODE45 is this algorithm! Error computation: $\Delta_{n+1} = \frac{|[\tilde{X}*n+1]-[X*n+1]|}{h}$. The error estimate is then compared to the error threshold ϵ_t provided by the user. If $\Delta_{n+1} \leq \epsilon_t$, the local truncation error is assumed to be smaller than the threshold, the state $[X_{n+1}]$ is accepted and the algorithm moves one step forward. In the other case, the new state is not accepted and the next state is evaluated again using a different (smaller) value of h. In both cases, the value of h is updated as:

$$h_{n+1} = h_n \sigma\sigma = (\frac{\epsilon_t}{2\Delta_{n+1}})^{1/4}$$

Issue: it's possible that for specific dynamics we will need an infinitesimal h to satisfy the error threshold. We can avoid this by updating the strategy, for instance impose an additional threshold to h or insert an heuristic.

The value of h used at each step is plotted in Figure 5.3. We can end up in situations in which the system changes a lot → *stiffness condition*, if we work with an adaptive method the property of the dynamics leads to instability in deriving the discretization step. Stiffness is a couple property of ODEs and the numerical scheme used to solve the system. This means that the same system of ODEs may exhibit stiffness only when it is simulated with some of the numerical schemes introduced in this chapter. In MATLAB we can use ODE15S in case of stiffness (happens quite often in biology simulations).

Chapter 6

Hybrid simulation approaches

Hybrid simulation combines the advantages of complementary simulation approaches: a system is partitioned into subsystems that are simulated with different methods. Sometimes we need to make a consistent choice for the complete model, but often (especially in biology) the model can incorporate many different situations. For instance, not all species will be present in low or high amounts. We cannot simulate with exact simulation strategies due to complexity, but at the same time for some parts of the network relying on approximation is too much and we risk losing some important details. We need to identify the qualities of the system that should be linked to a specific degree of approximation. When is it particularly dangerous to simulate deterministically? For instance, when rare events can occur: the stochastic nature of reaction firing cannot be fully modelled by deterministic setting. If the firing probability is lower than a threshold, we can assume that it is rare → intrinsically stochastic. We can divide reactions according to their firing probability, apply a rule over the propensity. If instead we focus on variables, we can select abundance levels: in case of low abundance, the stochasticity over the average is providing an important part of the behaviour. Usually, reactions with low propensity are also the ones modifying low abundance species. In highly abundant species, the contribution of noise is reduced.

- low number species and slow reactions → exact stochastic simulation
- high numbered species and fast reactions → approximation

In the scheme in Figure 6.1 we are trying to map the algorithms that we have already seen together, we are still not creating an hybrid method. Keep in mind that the availability of species can change along the trajectory. It is particularly difficult to make a consistent choice, therefore the hybrid setting is generally preferable in this case.
We need to partition the set of chemical reactions in subgroups in which we find consistent properties (Figure 6.2). Once the groups are established, we should be able to simulate the system in different settings. The main assumption in the tau leaping method is that the propensity of the reaction will not change dramatically during the macrostep; if we apply this concept to the hybrid setting we will see that this is an issue, as groups are not disjoined.

6.1 Reaction-Based System Partitioning

In order to divide reactions into group we can set up a threshold, which could be computed over the product of the propensity of the state and the simulation step. If the product is higher or lower than

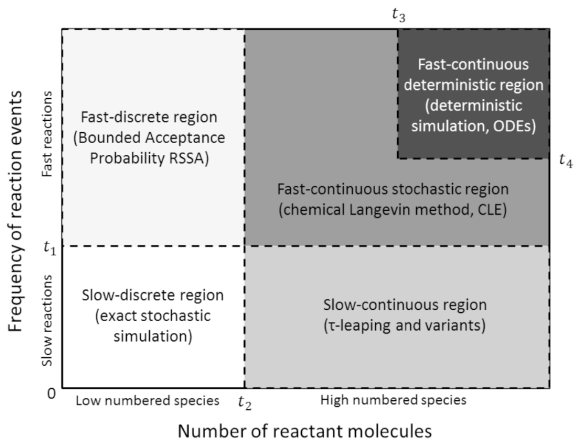

Figure 6.1

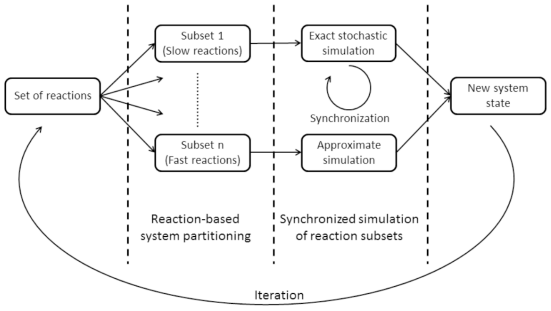

Figure 6.2

the threshold, we can identify the reaction as rare or probable. Example of partitioning algorithm: *two class reaction-based partitioning*. Divide reactions in slow and fast through an iterative loop. In general, we can increase the complexity of such approach as much as we want. Example: *four class reaction based partitioning*, we can bridge more simulation strategies. In Algorithm 45 the four class partitioning is applied; at the very beginning we impose a time step e.g. with tau leaping, compute the partitioning ending up with 4 sets (very slow, slow, medium and fast). For any of the reactions we will apply different strategies, for instance in the case of the very slow we require exact stochastic simulation. For sure the simulation will be more accurate, but we cannot claim that the simulation is exact, since we are only working on a set of reactions, not on the full system. If we wish to have an exact algorithm, we should consider the problem of **time varying reaction propensity**. If we want to be able to appropriately generate time, we should move considering the integral of the propensity over the time and the random number. We can consider the zero crossing of an equation as following (the log of the random number is a negative quantity):

$$\int_t^{t+\tau} a_0^s(X(t'))dt' = -\ln(r)$$

Instead of deciding τ at the beginning, the time will be computed along the approximation; when the quantity will be equal to zero, the approximation will stop and restart. We can consider this as a sort of traffic light: an event is generated, green light, we can move one. The real issue is that even if we have an equation to find the right time, being able to compute this requirement exactly with a computational strategy is a problem. The approach to zero will be affected by a variety of steps, e.g. the computer has a certain threshold for the zero. In addition computing the integral is computationally challenging.

1. non trivial complexity, integrals tend to be approximated by computers
2. the zero crossing is affected by approximation error

If we use deterministic simulation for simulating fast reactions, we can add another equation to understand when it is the time to stop during the simulation. We start from the logarithm of the random number and proceed with numerical integration.

$$\frac{dRES}{t} = a_0^x, RES(0) = \ln(r)$$

Synchronization has a price: the more complex, the higher impact we will have on the right time. Can we do something better? We can apply an extension of RSSA to obtain better results.

6.2 HRSSA

This algorithm claims to be exact and was developed by Marchetti. In RSSA we have a side effect: τ is computed over an *upper bound*, therefore we no longer need to reason in terms of propensity varying in time. By taking this perspective, we totally avoid slow events, just focus on upper bound. We have two main issues in this strategy:

1. by considering an upper bound we will generate more events
2. the bounds should be satisfied, therefore along the simulations we will need to check the consistency over the fluctuation interval of the state

Algorithm 47 Hybrid Rejection-based SSA (HRSSA)

Input: a biochemical reaction system with initial state X_0, the parameter δ for calculating the fluctuation interval of the system state, the parameters τ^f, θ and γ for running Algorithm 42 and the simulation ending time T_{max}.
Output: a trajectory of the biochemical system.

initialize time $t = 0$ and state $X = X_0$
while $(t < T_{max})$ **do**
  compute the fluctuation interval $[\underline{X}, \overline{X}] = [(1-\delta)X, (1+\delta)X]$
  for all reactions $R_j \in \mathscr{R}$ **do**
    compute reaction propensity bounds $\underline{a}_j$ and $\overline{a}_j$
    update reaction partitioning (sets $\mathscr{R}^s$ and $\mathscr{R}^f$) by applying Algorithm 42 on the lower bound of the system state $\underline{X}$ according to input parameters γ, θ and τ^f
  end for
  compute $\overline{a}_0^s = \sum_{R_j \in \mathscr{R}^s} \overline{a}_j$
  set $updateNeeded = false$
  while $(t < T_{max} \wedge \neg updateNeeded)$ **do**
    set $\tau = -ln(r)/\overline{a}_0^s$, where r is a random number in $\mathrm{U}(0,1)$ (see Appendix B.1)
    compute $X(t+\tau')$ by simulating fast reactions ($\mathscr{R}^f$), at time steps of maximum length τ^f, according to an approximate algorithm (either stochastic or deterministic), where $t+\tau'$ is the last computed time step such that $\tau' \leq \tau$ and $X(t+\tau') \in [\underline{X}, \overline{X}]$
    if $(\tau' = \tau)$ **then**
      generate two uniform random numbers $r_1, r_2 \sim \mathrm{U}(0,1)$
      set $accepted = false$
      select the slow reaction $R_\mu \in \mathscr{R}^s$ with the smallest index μ such that $\sum_{j=1}^{\mu} \overline{a}_j > r_1 \overline{a}_0^s$
      if $(r_2 \leq \underline{a}_\mu / \overline{a}_\mu)$ **then**
        update $accepted = true$
      else
        compute $a_\mu(X(t+\tau'))$
        if $(r_2 \leq a_\mu(X(t+\tau'))/\overline{a}_\mu)$ **then**
          update $accepted = true$
        end if
      end if
      if $(accepted = true)$ **then**
        update $X(t+\tau')$ by applying R_μ
      end if
      if $(X(t+\tau') \notin [\underline{X}, \overline{X}])$ **then**
        update $updateNeeded = true$
      end if
    else
      update $updateNeeded = true$
    end if
    update $X = X(t+\tau')$
    update $t = t+\tau'$
  end while
end while

Figure 6.3: HRSSA algorithm

We have no need of computing the integral, we work with fluctuation intervals and link the generation of τ over this quantity, with the only difference that we only consider the upper bound and not the current value.

Pseudocode (Figure 6.3): we start from a while loop, we compute the fluctuation interval, the bounds of the propensity (same as RSSA) and move forward with τ computation. When we reach τ we decide to apply or not the reaction through random number generation. The main issue is how to compute the reaction partitioning, in this case the algorithm divide into slow and fast. Since we are working with bounds, we substitute the real propensity with the lower bound for performing the partitioning. Given that the partitioning is computed on the bound, it is just necessary to compute it again for each interval $\rightarrow$ speed up. When the system is very complex, it is often not possible to apply this algorithm.

6.2.0.1 HSimulator

Simulator prototype (Java) developed by Marchetti to try HRSSA. We simply provide the set of reactions in arrow notation, same specification as MATLAB. The simulations provided are DM, RSSA, Euler, RK45 and HRSSA. We have the possibility to define simulation length and time sampling i.e. sampling over which the time series is stored.
Oregonator HRSSA: (Figure 6.4) the algorithm applies the stochastic approach by starting from the deterministic setting. In the advanced options we can insert additional specifications]. If we apply steady state conditions (simulation length = 5, time sampling = 0. 0001), we observe a

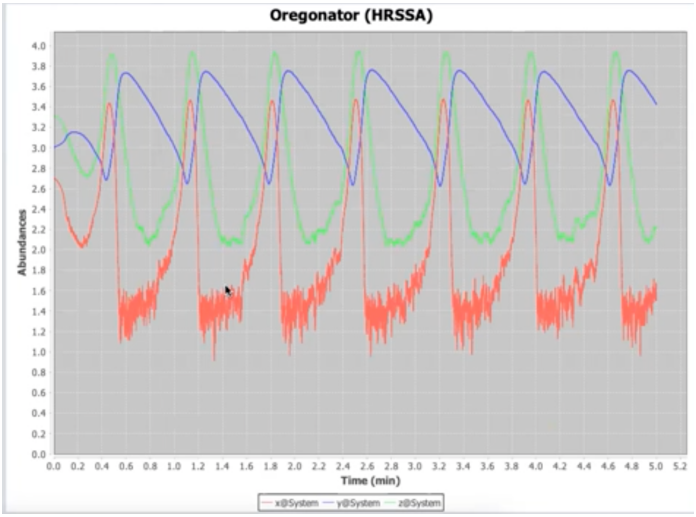

Figure 6.4: HSimulator HRSSA Oregonator

flat signal; the issue is that the stochastic simulation is applied to the subset of slow reactions. If we change the parameters for deciding if something is fast or slow, we should see a change in the behaviour. By working with smaller variables, we see something remarkable: noise is heavily present.

Chapter 7

Reali

7.1 Introduction

The course follows the structure of the article ***Optimization Algorithms for Computational Systems Biology.***

7.1.1 Definition of a system

A system is a set of integrated and interacting *components* or *entities* that form a whole with definite boundaries and surrounding environment. A system has a goal to achieve by performing one or more functions or tasks. Systems can be aggregated into a ***hierarchy***. A system at a given level of detail can be a component at a higher level of detail.

- A ***complex*** (non-linear) ***system*** is a system that does not satisfy the principle of superposition, i.e., the behavior of the system cannot be inferred from the behavior of its components.
- A ***dynamical system*** is a system where fixed rules define the time dependencies of the system in a geometrical space. Dynamical systems have a space and time dimension because they change their characteristics over time. If we pick snapshots of the system at different time points, we observe different configurations of the system (data).

A ***configuration*** or state of the system refers to the current condition of the system and stores enough information to predict its next move. A state is characterized by the position of its components in a geometrical space and by the values of the attributes of its components (e.g., concentration or number of each elements involved). Systems change their state over time by changing the location of some of their components or changing the attributes of some of their components.

- ***steady state***: some of the attributes of the system are no longer changing in the future.
- ***transient state***: time needed to reach the steady state.

7.1.2 Determinism, nondeterminism, or stochasticity?

- ***Deterministic systems*** always react in the same way to the same set of stimuli. These systems are completely determined by the initial state and the input set. The essence of deterministic systems is that each event is causally related to previous events and choices are always resolved in the same way in the same context. When a system generates multiple

outcomes from the same input in different observations, the system is ***nondeterministic*** (we cannot predict the output from the input).

- ***Stochasticity*** is the quality of lacking any predictable order or plan and stochastic systems possess some inherent randomness. It is possible to transform a nondeterministic system into a stochastic one by attaching probabilities to the selection points so that we turn nondeterministic choices into probabilistic choices.

7.1.3 Computational complexity

Complexity arises when interacting components self-organize to form evolving structures that exhibit a hierarchy of emergent system properties. An ***emergent behavior*** can be originated by a collection of components that interact in the absence of a centralized point of control to produce something that has not been designed or programmed in the system construction or evolution. Example: internet, ant colonies, consciousness.

Computational complexity is the amount of resources, measured as a function of the size of the input, needed to execute an algorithm.

- Computational space complexity: the amount of memory needed;
- Computational time complexity: the number of instructions to be executed.

7.1.4 Definition of a model

A *representation* is a set of symbols used to convey information and knowledge about a system. It is either physical as a cell or an ecosystem, or artificial as a computer network or an economic market. An abstraction is a representation that ignores some aspects of a system which are not of interest for the current investigation.

A *model* is an abstraction of a system. A model has its own interacting components that are characterized by the attributes that we want to observe. The set of all the attributes in a model is the *experimental frame*.

- A *dynamic model* aims at predicting the behavior of the system in time/space through what if analysis. **What if analysis** investigates how a change in some attributes affects the behavior of the modeled system.
- A *computational model* is a model that can be manipulated by a computer to observe properties of the corresponding system.

7.1.5 Checking the validity of a model

Validity is a fundamental property of models and witnesses the capacity of a model of making good predictions. We need to assess the validity of a model before using it to predict the behavior of a system.

Assume that M is a model for the system S and $\underline{M}$ is the modeling process. Let $s(t)$ and $m(t)$ be the state of the system and of the model at time t, and f_s and f_m the state transition functions of the system and of the model, respectively. Finally, let $I_s(t)$ and $O_s(t)$ be the input and output of the system at time t. Similarly, we write $I_m(t)$ and $O_m(t)$ for the model.

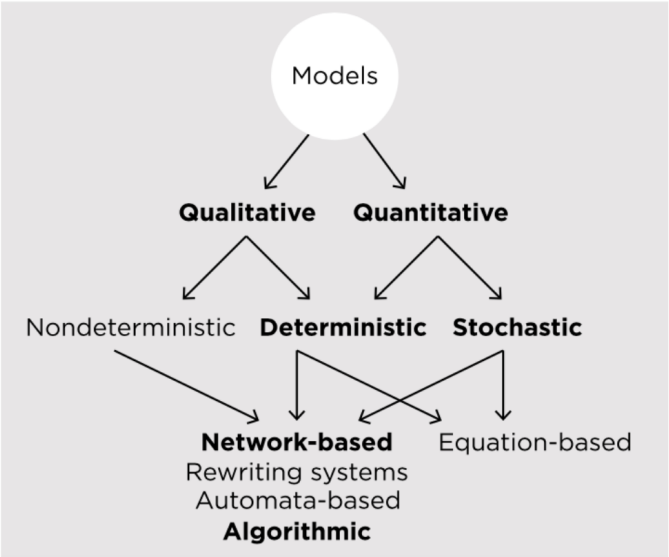

Figure 7.1: From a model to methods

What we expect is that going from one state to the other we have a function (one for the system and one for the model); in a mathematical model we integrate the f_m function to known what happens in the transition of the model, but we cannot do that in the real setting (the transition function f_s is not known) $\rightarrow$ when dealing with nature, we cannot validate models according to the previous definition, so we use I/O validity, based on known input and outputs of the system.

The input and output are here generalized concepts: input can be any perturbation of the system or of the model, and output can be any observable property causally related to the input.

A model M is valid for a system S if: $f_m(\underline{M}(s(t_0))) = \underline{M}(f_s(s(t_0))) = m(t_1)$
I/O validity can be checked by using data sets produced by the model and observed and measured on the system. An issue in this comparison process is *overfitting*:

- a model is well tuned to a specific dataset used to build the model
- it performs poorly on other datasets

Cross-validation: check overfitting by testing the model on data sets different from the ones used to build and calibrate/train the model.

These concepts, even if usually referred to computational models, may apply to general models or representations of a system.

7.1.6 How to build a model

We need to define objectives:

- what do you want to model?

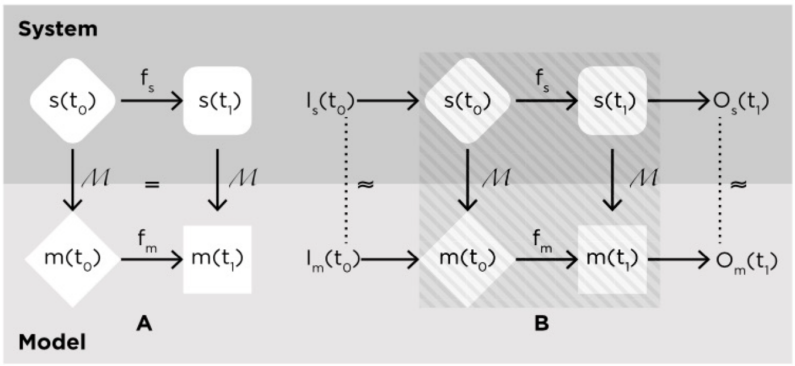

Figure 7.2: validation

- what do you want to investigate with the model?
- what do you expect from the model?
- why do you need a model?

After defining the question and gathering data, we need to build the model and calibrate it, in order to check if it can recapitulate data. If it does not, either we are missing something or we must tune some parameters. Different parameters can lead to dramatic changes in dynamics. Example: Lotka-Volterra model with different parameter conditions (Figure 7.4).

A) shows periodic oscillations, same amount of preys and predators

B) wider peaks and lower predator presence

7.2 Optimization problem

In general, it is a problem in which we try to *maximize* or *minimize* something. What we want to optimize is a function usually called *objective function* (or cost function). The function depends on a variable or a vector of variables, called unknowns or parameters or parameter estimates. They may be subject to certain constraints(<,>,=).

7.2.1 General definition of an optimization problem

$$\left\{ \begin{array}{ll} \max_{x \in \mathbb{R}^n} f(x) & \\ c_i(x) = 0 & i = \mathcal{E} \\ f_j(x) \geq 0 & j = I \end{array} \right\} \text{ set of indexes } \Bigg] \text{ Constraints (equality and inequality)}$$

The model (for us) is a function that gives a certain interval/time, initial conditions and parameters, returns the variables at that time. (Assume deterministic description)

$m : \mathbb{R} \times \mathbb{R}^{(n+1)} \times \mathbb{R}^m \to \mathbb{R}^N$

where n+1 accounts for the dimension of variables and time.

$$(t_1, (\vec{x}_0, t_0), \theta) \longmapsto \vec{x}_1$$

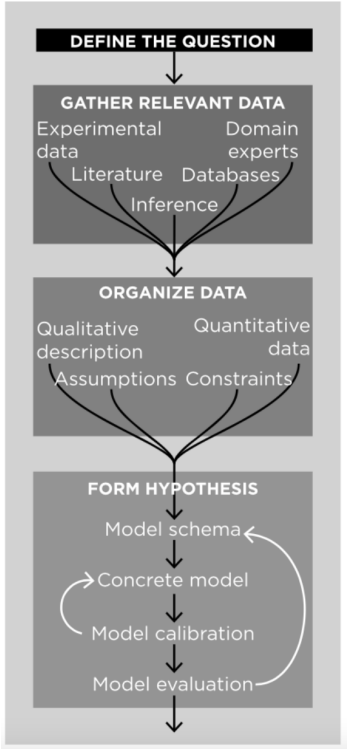

Figure 7.3: Workflow

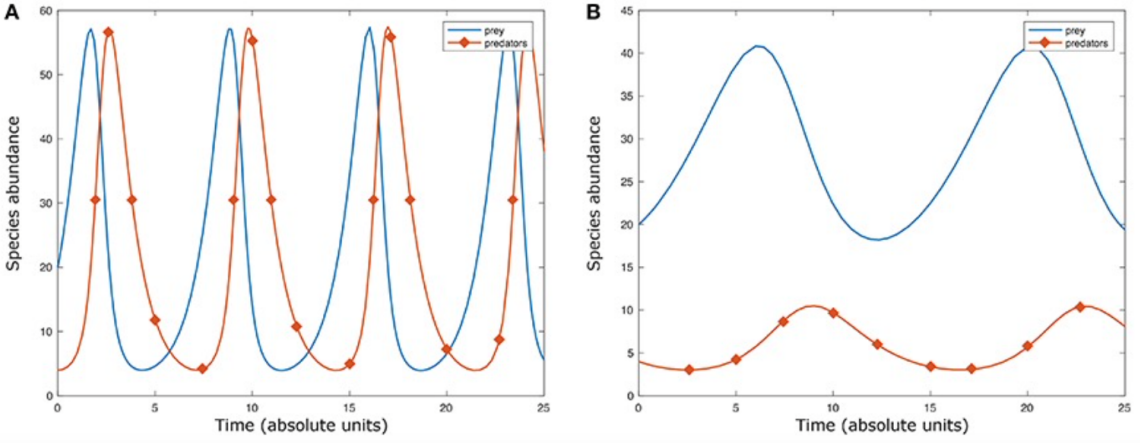

Figure 7.4: Volterra

Initial conditions in Lotka-Volterra: $((20, 5), 0)$.

θ represents parameters e.g. in Lotka-Volterra, a, b, β, α. Now, assume that we have k observations:

$(t_i, \vec{y}_i), i = 1, \dots k$, where $t_i \in \mathbb{R}^+, \vec{y}_i \in \mathbb{R}^\ell$ y in theory can be a subset, $\ell < N$: this happens a lot in complex systems, we may not observe all variables! For simplicity, we can assume $\ell = N$. Assuming that we can compute the following:

$$m\left(t_1, (\vec{x}_0, t_0), \theta\right) \in \mathbb{R}^N =_{\text{drop initial point notation}} m(t_i, \theta)$$

We can compute distance: $d_i = \vec{y}_i - m(t_i, \theta)$. We can choose any type of distance (Euclidean, max...). What we do is calculating point-wise the distance between the model and 'true' labels. It is quite common to use the ***Euclidean distance***:

$$d\varepsilon = \sqrt{\sum_{i=1}^{k} (\vec{y}_i, -m(t_i, \theta))^2} \to d\varepsilon - \sqrt{\sum_{i=1}^{K} \sum_{j=1}^{N} \left(y_{ij} - m_j\left(t_i, \theta\right)\right)^2}$$

Sometimes we need need to add weights, which multiply each component in the distances. We are putting together many outputs from the same model, so we might want to scale everything to make it more comparable. Furthermore, variables might be in different units of measurement, leading to biased results.

Observations:

- we do not want to reach "zero" when minimizing. Indeed, if the residual error = 0, we are 100% sure that we are overfitting the data.
- we need to really understand the data to construct the model
- we are manipulating θ in the space of the parameters, but we modify the output in the space of the observations: we are connecting abstract values to observations - like parameters for maximum likelihood.

Weights are multiplicative factors, sometimes we might wish to *transform* the distance.

Example: **Least squares algorithm**

$$d\varepsilon = \sqrt{\sum \sum W_{ij} \left(y_{ij} - m_j\left(t_i, \theta\right)\right)}$$

Our problem is to minimize/maximize a function. Assume:

$$\min f(x), x \in \mathbb{R}^n$$

7.2.2 Definition of a minimum

A point x^* is called **minimum** if $\exists \varepsilon > 0 : \forall x : ||x - x^*|| < \varepsilon$

$$\Rightarrow f(x) \geq f(x^*)$$

[For the maximum $f(x) \leq f(x^*)$]

The minimum is ***global*** if $\forall x \in \mathbb{R}^n$ (or in our domain) $f(x) \geq f(x^*)$. In general it is not easy

to determine global minimum/maximum, especially if we have a lot of dimensions. To find minima or maxima, we should impose $f'(x) = 0$.

We call a ***stationary point***, a $\bar{x}$ s.t.$f'(\bar{x}) = 0$.

7.3 Gradient methods

When we integrate to find ODE solutions, we do not obtain a function as a solution, just points. In our optimization problem we do not know f and f'(only sometimes we do), so we are required to use *numerical approximation*. The idea of looking at f' and set $f' = 0$ is still at the base of gradient methods.

If the problem contains constraints, how do we solve it? In this case the problem is:

$$\begin{cases} \max f(x) \\ g_i(x) = 0, \quad i \in \mathcal{I} \\ x \in \mathbb{R}^N \end{cases}$$

Let $\mathcal{I} = 1, ..., m$. The traditional way to solve this problem is to translate this system to another function. The Lagrangian function is used to take into account the constraints.

7.3.0.1 Lagrangian function

We define the ***Lagrangian function*** as $L : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$ s.t.

$$L(x) = f(x) + \lambda g(x) = f(x) + \sum_{j=1}^{m} \lambda_j g_j(x)$$

7.3.1 Lagrangian Multipliers Theorem

If x^* is a stationary point for (Lagrangian function), then $\exists \lambda^*$ s.t. $(x^* \lambda^*)$ is a stationary point for L. It is a necessary condition (not sufficient, only one direction). This is a "bigger" problem, from $\mathbb{R}^N \to \mathbb{R}^N \times \mathbb{R}^m$. But still, I can search solutions using stationary points. We can generalize the idea to $g_i(x) \leq 0$, constraints

Remember that stationary points are not necessarily minima and maxima. We check whether a stationary point is a max/min through second derivations or evaluate the function in "other" points.

7.3.2 Definition of a gradient

Let $f : \mathbb{R}^N \to \mathbb{R}$ a differentiable function, we call gradient of f

$$\nabla f : \mathbb{R}^N \to \mathbb{R}^N \text{ sit. } \nabla f_i = \frac{\partial}{\partial x_i} f(x)_i \text{ and } \nabla f(x) = \begin{bmatrix} \frac{\partial}{\partial x_1} f(x) \\ \vdots \\ \frac{y}{\partial x_N} f(x) \end{bmatrix}$$

We look for points for which the derivative vanishes $x^* : \nabla f(x^*) = 0$

TRY at home:$f(x, y) = (1 - x)^2 + 100 \left(y - x^2\right)^2$

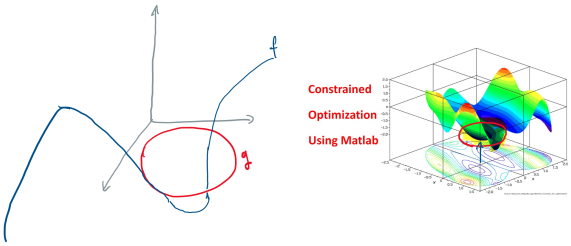

Figure 7.5: Blue = function, red = constraint

$$f(x,y) = -(y+47)\sin\left(\sqrt{\left|\frac{x}{2}+(y+47)\right|}\right) - x\cdot\sin\left(\sqrt{|\,x-(y+47)\,|}\right).$$

These two functions are used to test optimization algorithms. The first is **Rosenbrock's function,** the second the **Eggholder function**. Solving analytically these problems is hard. We cannot apply gradient methods for stochastic simulations, since the function is not continuous.

7.3.3 Limitations of gradient descent methods

One of the major limitations of these algorithms is that we are focusing on local minima, we never know if the distance is minimum. Furthermore, sometimes we want to optimize more variables and it might not be optimal to perfectly fit the solution to both of them → trade-off.

Figure 7.5: if we have an equality constraint we are only considering the points meeting the boundary (red). In an inequality constraint, we consider everything inside the red circle (yellow area). Generally, constraints reduce our search space; the Lagrangian tells us that the minimum point with some multipliers will give a solution of the Lagrangian, which is one function. If we find the solutions, we do not know if they are solutions of the original conditions, but they are ideal candidates that can then be checked.
For performing an evaluation of the distance we should integrate the model, which is computationally expensive. To do one integration we must perform a lot of computations. Our measure of computational cost is the number of times we have to simulate the model (per iteration).

In most cases, we do not know the gradient, therefore we should approximate it using the Taylor formula.

7.3.4 Gradient approximation with Taylor formula

$(a,b) \in \mathbb{R}, x_0 \in (a,b)$

Let $f_i(a,b) \to \mathbb{R}$ be differentiable $(n-1)$ times in (a,b) and $f^{(n)}$ is continuous in x_0. Then let $x \in (a,b)$, we have:

$$f(x) = f(x_0) + f'(x_0)(x-x_0) + f''(x_0)\frac{(x-x_0)^2}{2!} + \ldots + f^{(n)}(x_0)\frac{(x-x_0)^n}{n!} + R_n(x) \text{ s.t. } \lim_{x\to x_0}\frac{R_n(x)}{(x-x_0)^n} = 0$$

We focus on the first terms $f(x) = f(x_0) + f'(x_0)(x-x_0) + R_2(x) = 0$
$f'(x_0) = \frac{f(x)-f(x_0)}{x-x_0} + \left(\frac{R_2(x)}{(x-x_0)}\right) = \frac{f(x)-f(x_0)}{x-x_0} + R_1(x).$

We can also use this trick for $N > 1$

Let $f : \mathbb{R}^N \to \mathbb{R}$ and $e_i = (0, \dots, 0, 1, 0, \dots, 0)$. Let's consider $x_1 x + \varepsilon e_i, x - \varepsilon e_i$; we are only moving along one direction. In this case:

$$f(x + \varepsilon e_i) = f(x) + \varepsilon \frac{\partial f}{\partial x_i}(x) + \frac{1}{2}\varepsilon^2 \frac{\partial^2 f}{\partial {x_i}^2}(x) + R_3(x) f(x - \varepsilon e_i) = f(x) - \varepsilon \frac{\partial f}{\partial x_i}(x) + \frac{1}{2}\varepsilon^2 \frac{\partial^2 f}{\partial {x_i}^2}(x) + R_3(x) \Rightarrow f(x + \varepsilon e_i) - f(x$$

We have computed an approximation of the first derivative with improved accuracy.

Consider that this only applies to one derivative, we have to perform this at least twice $\to$ 2 function evaluation for each $i \Rightarrow 2W$. In order to obtain a decent gradient, we require a lot of computations, but they are fast (quite low number of iterations). We will see other methods, which are somehow more precise, but also heavier.

We can always check $\Delta f = 0$ or not to understand if we are done!

As we already saw, there might be points where the gradient vanishes which are not the final destination. Gradient methods may tend to overfitting, but they are effective. The main issue is that since we approximate the gradient, we do not trust it everywhere.

Gradient methods can be applied to two different categories of problems:

- constrained
- unconstrained
  - **line search algorithm**: follow a direction
  - **trust region:** create an approximation of the problem and solve it in a small trustable region

7.3.5 Line search

7.3.5.1 Newton's direction

If we consider Taylor's formula and let x_k be our starting point, let $\alpha \in \mathbb{R}^+$ step length and p our direction ($x_k, p \in R^n$). For $n = 1$, we have:

$$f(x_k + \alpha p) = f(x_k) + \alpha p f'(x_k) + \frac{\alpha^2 p^2}{2} f''(x_k) + r(p^3)$$

For simplicity set $\alpha = 1$ and truncate the formula:

$$f(x_n + p) = f(x_n) + p f'(x_n) + \frac{p^2}{2} f''(x_n) = m_k(p)$$

$\to$ instead of minimizing the initial f, we minimize the simple polynomial $m_k(p)$. From this equation we can get the direction $p = -\frac{f'(x_k)}{f''(x_k)}$, which is called *Newton direction* (best direction). When $n > 1$, $p = -(\nabla^2 f(x_k))^{-1} \nabla f(x_k)$. Finding the inverse of a matrix is "a pain", not straightforward; this is why we will try to approximate this part.

7.3.5.2 Line search algorithm for function minimization

Set $k = 0$ and guess an initial point x_0 WHILE $||\nabla f(x_k)|| > 0$

1. compute $p_k = -(\nabla^2 f(x_k))^{-1} \nabla f(x_k)$
2. select α_k
3. update $x_{k+1} = x_k + \alpha_k p_k = x_k - \alpha_k (\nabla^2 f(x_k))^{-1} \nabla f(x_k)$
4. $k = k + 1$

We stop when the gradient is equal to zero, but we do not really have zeroes in our computers; therefore, we must apply a threshold ε.

At each iteration we have some issues:

- sometimes we require approximation to compute the inverse of a matrix
- it is computationally expensive to compute all these gradients

Nevertheless, going in this direction is smart. We can try to tackle the hardest part, i.e. p_k computation, by approximation.

7.3.5.3 Quasi Newton's direction

To avoid the computation of $\nabla^2 f$, we build iteratively a different matrix B_k, such that $B_{k+1} \cdot (x_{k+1} - x_k) = \nabla f(x_{k+1}) - \nabla f(x_k)$ [condition for the matrix]

We are approximating the second derivative as second of the gradient. Since we are already computing the gradient, we exploit it to prevent extra computations. The difference with respect to standard Newton direction is the Hessian, which is not computed here.

Quasi-Newton direction: $P_{k+1} = -\alpha B_{k+1}^{-1} \nabla f(x_k + 1)$

7.3.5.4 Steepest descent direction

We proceed in the direction that reduces the gradient the most:

$$p = -\alpha \frac{\nabla f}{||\nabla f||}$$

In this case we are completely ignoring the second derivative, we only focus on the gradient computation. This method converges slowly with respect to Newton and Quasi-Newton's direction.

7.3.5.5 Selecting α

Ideally, we want to define α s.t.

$$\bar{\alpha} = \arg\min_{\alpha > 0} f(x_k + \alpha p)$$

This is another optimization problem, ideal situation. To avoid solving the optimization, we usually consider α that satisfies :

$$f(x_k + \alpha p) \leq f(x_k) + c_1 \alpha \nabla f(x_n)^T p_k, c_1 \in (0, 1)$$

Armijo condition: the reduction should be proportional to α and $\nabla f(x_n)^T p_k$. Sometimes thing fails because of parameter choice: it is crucial to understand why functions fails and how parameters tuning can affect the most crucial steps in the algorithms.

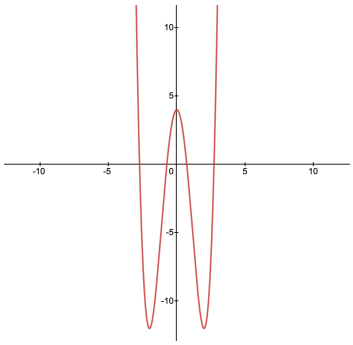

Figure 7.6: Desmos $f(x) = x^4 - 8x^2 + 4$

7.3.5.6 Convergence of a method

The order of convergence of a method (for us) is a constant ℓ, such that exists the limit:

$$\lim_{k\to\infty} \frac{||f(x_{k+1}) - f(x^*)||}{||f(x_k) - f(x^*)||^\ell} = L > 0$$

Where x^* is the solution and the method converges.

We compare the new iteration (numerator) to the old iteration (denominator); the limit should go to zero, the exponent is the speed - the higher the exponent ℓ, the faster will the upper term go to zero with respect to the bottom one i.e. faster convergence. The higher the exponent, the less iterations are required (in general). For the methods that we have previously seen:

- steepest descent: $\ell = 1$, linear convergence
- quasi-Newton: $\ell \in (1, 2)$, superlinear convergence
- Newton: $\ell = 2$, quadratic convergence

Example: $f(x) = x^4 - 8x^2 + 4$

$$x_0 = 3f'(x) = 4x^3 - 16x \Rightarrow f'(x) = 0 \Rightarrow x = 0, x = \pm 2f''(x) = 12x^2 - 16$$

Algorithm: $f'(3) = 60, f''(3) = 92$,
Let $\alpha = 1$:

$$p_0 = -\frac{60}{92} = -0.65x_1 = 3-0.65 = 2.35f'(2.35) = 14.31f''(2.35) = 50.27p_1 = -\frac{f'(x_1)}{f''(x_1)} = -0.28x_2 = x_1+\alpha p_1 = 2.35-0.$$

Let's try the same by applying Steepest Descent method:
$f'(3) = 60$
$x_1 = x_0$.. $f'(3) = 60$ In this case the SDM is way faster, lucky shot. But what if we change the starting point? The direction will always be the same, i.e. $p_1 = -1$.

$$x_1 = 2.35 - 1 = 1.35f'(x_1) = -11.75x_2 = 1.35 + 1 = 2.35$$

We are doing ping-pong among two points!

The fact that the second derivative progressively shrinks tells us that we need to reduce the step, but in this case we are not taking this into account. Of course we also have α, we should look at Armijo condition and change it. Take into account that each time that we are performing operation on a number we lose precision.

7.3.6 Trust region

[picture] Imagine that this is a cut function: our starting point x_k is in the middle. The main idea of the trust region is that we do not follow a direction: we approximate the function with a simpler function $\rightarrow$ Taylor approximation. According to how big and reliable the approximation is, we will choose a direction.

Last time we approximated the model as:

$$m(x_p + \alpha p) = f(x_k) + \alpha p^T \nabla f(x_k) + \frac{1}{2}\alpha^2 p^T B_k p$$

$$\alpha = 1, m_k(p) = f_k + p^T \nabla f_k + \frac{1}{2} p^T B_k p$$

7.3.6.1 Trust region steepest descent

Define a region such that $||p|| < \delta_k, \delta_k > 0$ in which we solve the optimization problem (1) instead of the original. Remember that B_k can be the Hessian or an approximation; on the other hand, we have said that we can also ignore it.

Finding a minimum for $m_k(p) = f_k + p^T \nabla f_k$ means that we are looking for:

$$\min_p m_k(p) = \min_p (f_k + p^T \nabla f_k)$$

Remember that f_k is a constant, so we want to find a direction for which $\min_p p^T \nabla f_k$ is minimum. We can rewrite this as:

$$p^T \nabla f_k = ||p||||\nabla f_k|| \cos\theta$$

We minimize for p such that $\cos\theta = -1$ and $||p|| = \delta_k$, where p s.t. $||p|| \leq \delta_k$ [radius of the trust region].

$$\min_p p^T \nabla f_k = -\delta_k ||\nabla f_k||$$

$$p = -\delta_k \frac{\nabla f_k}{||\nabla f_k||}$$

This result is exactly the equation from steepest descent. We are applying a condition on the region with δ_k. This direction and the whole approach is called trust region steepest descent.

We could follow the same idea by applying Newton or Quasi-Newton.

To evaluate if we can really trust our region, we define the ***actual reduction*** as:

$$\rho_k = \frac{f(x_k) - f(x_k + p)}{m(x_k) - m(x_k + p)}$$

By construction $m(x_k + p) \leq m(x_k)$.

- If $\rho_k < 0 \rightarrow$ reject p, we are not improving the real problem. Usually take $\delta_k = \frac{1}{4}\delta_k$
- If $\rho_k \simeq 1 \rightarrow$ maybe longer step

δ value can be tuned according to the needs. The approach is similar to RK method seen with Marchetti. Of course we have a grey area between 0 and 1, so we define a threshold e.g. $\rho_k < \eta$ and $\rho_k > \eta$. By default, MATLAB uses a trust region algorithm. Having something certifying that we are doing good or bad is a great thing in approximation!

7.3.6.2 Trust region algorithm

Let $\hat{\delta} > 0, \delta_0 \in (0, \hat{\delta}), \eta \in (0, \frac{1}{4})$
$k = 0, \varepsilon < 0$
REPEAT
obtain p_k s.t. $p_k = \arg\min_p m(x_k + p)$
compute ρ_k
IF $\rho_k < \frac{1}{4}$
$\delta_{k+1} = \frac{1}{4}\delta_k$
ELSEIF $(\rho_k < \frac{3}{4})$ AND $||p_k|| = \delta_k$
$\delta_{k+1} = \min(2\delta_k, \hat{\delta})$
ELSE
$\delta_{k+1} = \delta_k$
IF $\rho_k > \eta$
$x_{k+1} = x_k + p_k$
ELSE
$x_{k+1} = x_k$
IF $||\nabla f(x_{k+1})|| < 0$
BREAK

We stop if the gradient is sufficiently small. Our focus is on computing p_k. We then evaluate ρ_k to adjust parameters (which is simpler from what we have previously seen).

If we apply this algorithm to the example of last lecture, it happens that with $x_0 = 235$ and steepest descent $\rho_0 = 0.17 < \frac{1}{4}$, which tells us to reduce δ. The only thing that changes is ρ changes.

PROBLEM:

$$g(x, y) = x^4 + y^4 + xy$$

`https://www.benfrederickson.com/numerical-optimization/` play with learning rate, explore the site.

7.4 Least squares problems

Recall that if m is the model, we quantified the distance between the model output and data as:

$$m_j(t_i, (x_0, t_0), \theta) = m_{ij}(\theta) - \hat{y}_{ij}, \text{for } i, j \text{ as lesson 2}$$

$r_{ij}(\theta)$ is called RESIDUAL and we shape it as a vector.

$$J_k = J(\theta_k)[\frac{\partial r_I}{\partial \theta_i}], i = 1, ..., n, I = 1, ...n$$

Since the time points are given by the data, everything only depends on the choice of parameter theta, which is a vector of parameters. We can derive the residual according to theta. Our function to minimize is $f(\theta) = \frac{1}{2}\sum_{j=1}^{m} r_j^2(\theta)$ It will hardly go to zero, as our observations are affected by noise: we just need to explain data, not noise. If we have the distance of the single point we can see the effect of each parameter by looking at the derivative. Here we have that the gradient (Jacobian) will be telling us the relationship among model parameters and data.

$$\nabla f(\theta) = \sum_{j=1}^{m} r_j(\theta)\nabla r_j(\theta) = J(\theta)^T r(\theta)$$

The matrix notation is a more convenient way to express this gradient. While solving least squares problems we always exploit Taylor approximation.

Hessian matrix:

$$\nabla^2 f(\theta) = J^T(\theta)J(\theta) + \sum_{j=1}^{m} r_j(\theta)\nabla^2 r_j(\theta)$$

What's *magical* of this is that we can use $J^T(\theta)J(\theta)$ as approximation for $B(\theta)$ of our gradient.

If the problem is linear, $r(\theta) = A(\theta) - y$. The objective function $f(\theta) = \frac{1}{2}||A\theta - y||^2$ and $\nabla f(\theta) = A^T(A\theta - y), \nabla^2 f(\theta) = A^T A$.

If f is convex $\Rightarrow \exists \theta^* s.t. \nabla f(\theta^*) = 0 - A^T A\theta^* = A^T y$

We reach a normal equation, linear system (we know how to solve this). With general functions, this is not so straightforward; what we will do is approximating the problem with a solvable linear problem.

When we try to quantify the distance between the model and our points we can formalize the problem as:

$$r_i = m_i(t, \theta) - y_0 f(\theta) = \frac{1}{2}\sum_{j=1}^{m} r_j^2(\theta)\nabla f(\theta) = J(\theta)^T r(\theta)\nabla f(\theta) = J(\theta)^T J(\theta) + \sum_{j=1}^{m} r_j \theta \nabla^2 r_3 \theta$$

Last time we already said that we will ignore the sum, for two reasons:

1. it contains second order derivatives, painful to compute
2. the Newton direction is a quasi vector

So at every iteration k we solve the approximated problem

$$J(\theta_k)^T J(\theta_k)p = -J(\theta_k)r(\theta_k)J_k^T J_k p = -J_k^T r_k$$

This is a linear system which we can solve. In this case, we are solving:

$$f(\theta_k + p) = \frac{1}{2}||r(\theta_k + p)||^2 \simeq \frac{1}{2}||J_k p + r_k||^2$$

This is what we called normal equation for a normal least squares problem. Under certain hypotheses the method converges quadratically!

We discussed last time that Newton is quadratic convergent. Of course the matrix should not be singular, we need to be able to solve the system. We start with a problem in a special form, sum of squares. Thanks to this, we can rewrite the problem in a simpler way and perform an approximation, leading to solving a linear system at each iteration. This approximation guides us rapidly to a solution. We are applying linear search, by defining a direction and solving a new problem at each iteration. The biggest issue could be the non invertible matrix, but we can do something to circumvent the issue.

7.4.1 The Levenberg-Marquardt method

At each iteration, we solve the problem:

$$\min_{||p|| \leq \delta_k} \frac{1}{2}||J_k p - r_k||^2$$

We can have a solution inside the trust region δ_k or on the border i.e. not the minimum, but the smallest value we can reach.

- case 1: β is a solution and $||p|| \leq \delta_k \rightarrow$ DONE
- case 2: $||p|| = \delta_k$, then $\bar{p}$ is a solution if and only if $\exists \lambda > 0$:

$$(J^T J - \lambda I) = -J^T r + \lambda(\delta_k - ||p||) = 0$$

If we can push it a bit further from zero and the problem is still solved, we can think of it as a solution. If we are hitting the boundary $\lambda(\delta_k - ||p||) = 0$, we want to move the matrix a bit from singularity. We will not use it very much, but it is a well known algorithm. It is one of the default MATLAB solvers. The Levenberg-Marquardt converges quadratically when we are close to the solution, while if the residuals are big it does not perform well.

7.4.2 Solving a problem with bounds

We have seen together that we can exploit the Lagrangian for the equality constraint, but we can have other boundaries. There are other approaches which allow us not to lose our approximation advantage. We can simply do ***variable transformation*** : instead of changing the problem, we change the variables. Example: $x \rightarrow e^x, \mathbb{R} \rightarrow \mathbb{R}_0^?$ [other example missing, cannot understand from recording]

Another solution could be to include the bound in the ***trust region***.

[drawing missing] *Evolutionary algorithms* follow the evolution of the solution: at the beginning look for solutions, then add a penalty for the boundaries.

7.4.3 Solving global minimum problem

How can we be sure that we are looking at a global minimum? We can start from another initial point and check whether the solution reaches the same minimum. From local to global → the ***multi-start approach,*** t is a shortcut, but the best way to work.

7.4.4 Gauss-Newton method

Let $N \in \mathbb{N}, \varepsilon > 0, J$ as defined before. We randomly select N vectors and use $\theta^1, \dots, \theta^n \in \mathbb{R}^n$
For each $i = 1, \dots, N$ DO
REPEAT
$\bar{J} = J(\theta^i)$
compute q s.t. $\bar{J}^i \bar{J} q = -\bar{J} r(\theta^i)$
compute $\vartheta = \theta^i + q$
compute $\varepsilon = ||\frac{\theta^i - \vartheta}{\theta}||$, *relative increase*
update $\theta^i = \vartheta$
IF $\varepsilon < \bar{\varepsilon}$ then BREAK
SAVE θ^i
END
→ no global minimum guarantee!

There are other ways to detect if a minimum is global e.g. we try to divide the space in N different sectors and pick points for each one of them, *Latin hypercube* and *orthogonal sampling.* The more we do it, the more we acquire confidence, but we will never be sure. In addition, we always have to deal with noise, we do not know if the minimum is the best or we are overfitting.

Example on MATLAB: if we take only 10 random points we can have a biased result. By taking a bigger sample, we should somehow achieve a fuller result in the region. Still, we have some holes and repeated points. Are there smarter ways to take random points?

7.4.5 Latin hypercube sample

Get random numbers dividing the interval in m sectors. This is a 2x2 problem and we need to reason on some aspects: if we take the square (0,1),(0,1) out of 100 points we will just have 3/4 there → not exploring it very well. If parameters are between 0.1 to 1000 we need to better sample.
$n = 20$, the fewer points we get, the more we see the difference among the two methods.
$n = 200$, still we have missing parts.
$n = 2000$, thousands of iterations → if we zoom in there is still a lot of space

The takehome message is that it is almost impossible to cover all the space with points. Luckily these are starting points, if they are close to the solution we will have the good direction. Gradient methods are expensive, but they are smart.

7.4.6 MATLAB

When an optimizer stops, it gives as output the function converged to the solution or why it stopped. Termination criteria: gradient, incremental step e.g. is our step too small?, change in x was less of certain tolerance, the residual was less than specific tolerance, max number of iterations. These conditions are added in order to avoid infinite function evaluation.

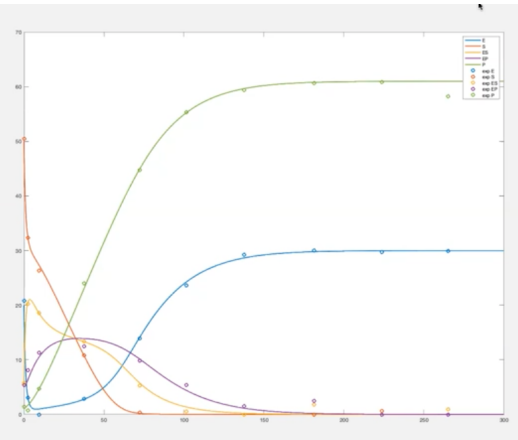

Figure 7.7: MATLAB multi-start lsqnonlin

We first need to define an enzymatic reaction as a set of ODEs. We then specify initial concentrations and a given set of rates. Integrate the model and plot the simulation results.

The dots are the experimental points, lines are the output of the simulation. According to different starting points, we will obtain different results; we need to quantify how much the model is distant from the real data. The output of the normal simulation is a set of points; we can do a linear interpolation for time and values, giving how much is each of the simulated variables at the requested time. Once we have the values at the right time, we can compute the *residuals* through the sum of squares. In some cases we can also look at weighted residuals and normalize them; however, zeros are problematic → our experimental data goes to zero, we stick to normal SSE.

Our objective function takes the rate, experimental time and values, initial and final time. We call the solver (15s as the system is stiff, we require a more robust iteration) and get a vector of residuals and SSE.

If we compute the residual with s_1,s_2 and s_3 we can easily understand which is the better one (matches visual inspection of the plot).

We can then optimize and find the solution with `lsqnonlin` e.g. $SSE1 = 33.6653$, $resNorm = 38$, worse, we should improve initial conditions.

[Recap from last time]

MATLAB multi-start is a wrapper working with different algorithms. It will automatically parallelize the problem. It requires to insert starting points and tolerance. We therefore define bound, create problem and give initial points e.g. lsqnonlin + objective function. The output contains the result of parallel lsqnonlin and parameters, somehow similar to what we saw before. We can reduce the bounds, but solution 1 is still the best. Main limitation: heavily depends on the number of initial points.
Remarks:

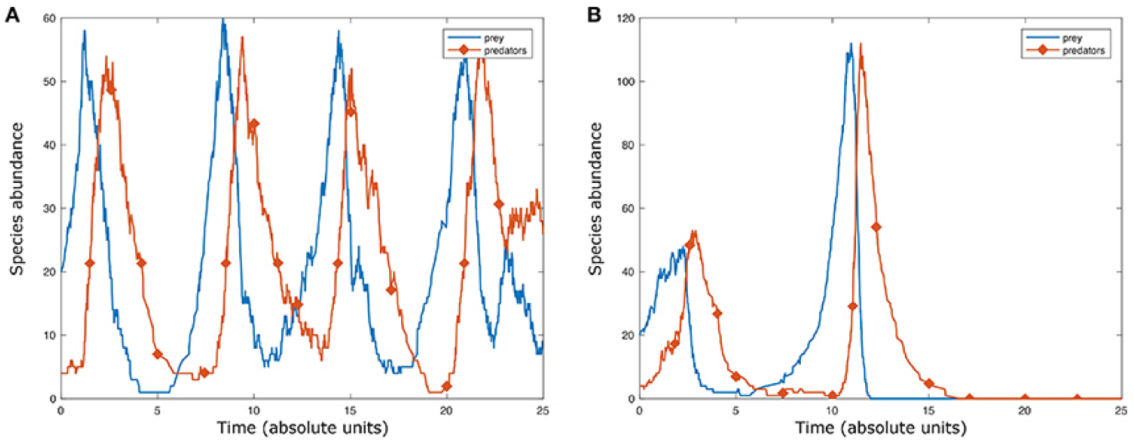

Figure 7.8: Lotka-Volterra stochastic simulation results. In B we witness preys and predators extinction, it is the output of a simulation performed with the same model and parameters as A.

- we did a lot of computations
- s_1, the initial set of parameters, had values at different orders of magnitude. We really do not know if we are doing well or not.

7.5 Stochastic methods for parameter estimation

Stochastic methods are often used when we cannot compute the gradient. With the same parameters, stochastic methods can produce dramatically different results.

Lotka-Volterra stochastic simulation results are reported in Figure 7.8- In B we witness preys and predators extinction, it is the output of a simulation performed with the same model and parameters as A.

Instead of following the gradient, we collect information in a manner resembling natural selection. Monte Carlo Methods are dated around 1949, the name comes from casinos (recalls concept of luck). In order to use MCM to do inference, Metropoli and Hastings developed a specific algorithm in 1970.

7.5.1 Markov Chain Monte Carlo (MCMC)

MCMC is a chain of events where the current state depends on the previous one and on the transition probability. If $x^{(i)}$ is a random variable (stochastic process) and it takes only discrete values $\{x_1, \ldots, x_s\}$. Let $p(x)$ be the probability distribution of x. $x^{(i)}$ is a Markov Chain if:

$$p(x^{(i)}|x^{(i-1)}, \ldots, x^{(1)}) = T(x^{(i)}|x^{(i-1)})$$

Simplest case: a MC is **homogenous** if $T(x^{(i)})x^{(i-1)} = T$, i.e. the transition matrix is constant.

If we run MCMC long enough, they will hopefully reach a stable point.

Example: 3 states and homogenous transition matrix

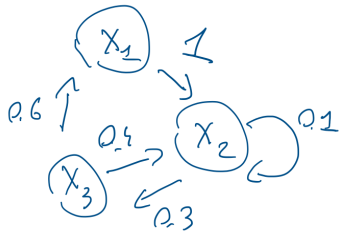

Figure 7.9: MCMC example

$$T = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0.1 & 0.9 \\ 0.6 & 0.4 & 0 \end{bmatrix}$$

MCMC example (Figure 7.9):

$$\pi_1 = \begin{pmatrix} 0.5 & 0.2 & 0.3 \end{pmatrix}$$

The next probability of being in the three states is given by

$$\pi_1 \cdot T = \begin{pmatrix} 0.3 \cdot 0.6, & 0.5 + 0.02 + 0.12, & 0.18 \end{pmatrix} = \begin{pmatrix} 0.18, & 0.64, & 0.18 \end{pmatrix}$$

If we do this enough, we will always arrive to a fixed distribution, called invariant distribution $\Rightarrow$ we want to build a Markov Chain whose invariant distribution is the distribution of our unknown parameters.

$$p(x) = \cdots = \begin{pmatrix} 0.2213 \\ 0.4098 \\ 0.3689 \end{pmatrix}^T$$

Where p(x) is called **invariant distribution**.

A stochastic matrix is *irreducible* if its graph does not contain unconnected sub-graphs. If T is an irreducible transition matrix (+ aperiodic) $\rightarrow$ the MC has an invariant distribution. We must connect probability distribution / values for the parameters with the model. There is a known function that we can use to do so.

Monte Carlo refers to a family of methods developed for **sampling**. In particular, we obtain samples of a certain i.i.d. random variables $\{x^{(i)}\}_{i>0}$ from a known density $p(X)$.

The samples can be used to "estimate / approximate" a target density / distribution. We exploit known distributions e.g. Normal, Poisson,... to approximate our unknown distribution of interest.

Given enough samples, we can approximate p as:

$$p_N(x) = \frac{1}{N} \sum_{i=1}^{N} \delta_{x^{(i)}}(x)$$

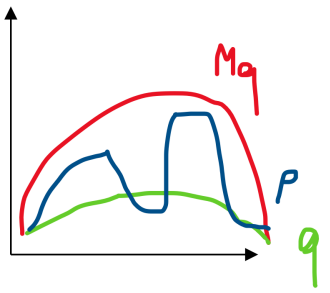

Figure 7.10: Example: bimodal distribution

with $\delta_{x^{(i)}}(x) = \begin{cases} 1, & x = x^{(i)} \\ 0, & \text{elsewhere} \end{cases}$ This is often used to approximate "hard" integrals. We can approximate and integral with a finite sum of values, which is obtained as:

$$I_N(f) = \frac{1}{N} \sum_{i=1}^{N} f(x^{(i)})$$

Here we are evaluating the function is some points. We can use the mean as an approximation of the integral. This equation converges to the real integral by the law of large numbers.

$$I_N(f) = \frac{1}{N} \sum_{i=1}^{N} f(x^{(i)}) \xrightarrow[N \to \infty]{} I(f) = \int_x f(x)p(x)dx$$

A probability is richer than the output of one least squares, we have information on the shape.

7.5.2 Sampling a distribution

Let p be a known probability distribution (up to a proportionality constant). We usually prefer to sample from a well-known distribution $q(x)$ s.t. $\exists M > 0$

$$p(x) \leq Mq(x)$$

We extract values from q if they are smaller than Mq; each time we are collecting information on p collecting from q.

7.5.3 Rejection sampling algorithm

Set $i = 1$
REPEAT
Sample $x^{(i)}$ from q and $u \in \mathcal{U}(0, 1)$
IF $u < \frac{p(x^{(i)})}{Mq(X^i)}$ and $x^{(i)}$
$i = i + 1$
ELSE
REJECT

If the point is in the area between p and Mq we still accept it; the further the ratio is from one, the smaller the chance of keeping the point.

At this point we are using a known distribution p; we then gradually remove what we know in the following iterations.

7.5.4 Metropolis Hastings

Let X be our current point, q the proposal distribution and p the target distribution. p can be an unnormalized density ($\int p > 1$ but $\int p < \infty$). Ideally we want to know p, but if it is unknown we can still proceed. Let us call this function h (non negative and positive integral).

X^* is the new candidate point. The trick of MH algorithm is to avoid relying on probability, we use a function. We require info on p, but not a probability.

Metropolis ratio $r_M(x, x^*) = \frac{h(x^*)}{h(x)}$

Hastings ratio $r_H(x, x^*) = \frac{h(x^*) \cdot q(x^*|x)}{h(x) \cdot q(x|x^*)}$ From the distribution, we want to measure how likely the previous value is based on the current value. We are weighting our choices to likeliness.

7.5.4.1 MH algorithm

Guess $x^{(i)}$
FOR $i = 0$ to $N - 1$
Sample x^* from $q(\cdot|x^{(i)})$ and $u \in \mathcal{U}(0, 1)$
IF $u < \min\{1, r_H(x, x^*)\}$
$x^{(i+1)} = x^{(*)}$
ELSE
$x^{(i+1)} = x^{(i)}$
END
END

We sample from a proposal distribution and accept according to another function i.e. if it is more likely. If it is not more likely, we accept it with a certain chance.

MATLAB plot:

- right: histogram with the known distribution
- left: MCMC oscillating between 0 and 10, recapitulates the distribution

The procedure is really fast, takes less than a second. If instead of extracting u and v from random distribution with an index, the result is the same → "thanks MATLAB, it is not necessary to prelocate anymore"

If we employ a Gaussian distribution, the result is still good but not as accurate as the previous one. We are adding complexity by introducing the variance; by choosing a different value we can improve the result. Pay attention to the fact that if we sample from a narrow distribution, we risk focusing only on one of the two points.

Everything we do has consequences!

7.5.4.2 Visualization of MCMC

`http://chi-feng.github.io/mcmc-demo/app.html?algorithm=RandomWalkMH&target=banana` Green: accept, red: reject

Target distribution = standard

- GibbsSampling: collects points according to a certain direction from a starting point, it tries to rebuild a 2D Normal distribution.
- AdaptiveMH: sample from a starting mean and accept or reject new points.
- Random walk: more or less like MH. Even if the target distribution is not that difficult (bell shape), there are a lot of rejections initially.
- DE-MCMC-Z: produces vectors.

Target distribution = donut

- SVGO: stochastic vector gradient descent, intermediate between gradient and stochastic.
- EfficientNUTS (No-U-Turn samples): it creates many points, more complex but one of the best. In the end it will converge quite fast.
- RandomWalk: needs more time to converge with respect to standard distribution.

Target distribution = multimodal

- RandomWalk: three initial points, it proposes a random perturbation.
- AdaptiveMH: changes some of the parameters. Differently from the previous approach, the shape changes: instead of having a fixed search area, it evolves and adapts.

7.5.4.3 How to link data and MH

In LSQ, we used the function

$$f(\theta) = \frac{1}{2}\sum_{j=1}^{m} r_j^2(\theta) = \frac{1}{2}\sum_{j=1}^{m}(y_j - m_j(t_i,\theta))^2$$

We can weight our residuals for their uncertainty:

$$f_w(\theta) = \frac{1}{2}\sum_{j=1}^{m}\frac{r_j^2(\theta)}{\vartheta_j^2}$$

ϑ_j= (for example) the standard deviation of that measured point.

We can use as function to link the parameters and the output, and be non-negative: [where Y is the vector of our observations]

$$L(\theta|Y) = \exp(-f_w(\theta)) \geq 0$$

We can use this as h in the MH algorithm.

Metropolis ratio:

$$r_H(\theta^*,\theta) = \frac{L(\theta^*)}{L(\theta)} = \frac{e^{-f_w(\theta^*)}}{e^{-f_w(\theta)}} = \exp(-f_w(\theta^*) + -f_w(\theta))$$

So since $f_w(\theta) \geq 0$ we have that if $f_w(\theta^*) > f_w(\theta) \Rightarrow r_H > 1$, we ACCEPT. Otherwise we accept with a certain probability. This kind of function can be used to link the new parameters with data. We can work with log-likelihood also here, we just look at $f_w(\theta^*)$ and $f_w(\theta)$.

7.5.5 Random walk MCMC

Let us assume θ^* is sampled from $\mathcal{N}(0, \mathcal{C}) + \theta = \mathcal{N}(\theta, \mathcal{C})$

We add a perturbation with $\mathcal{C}$, a known covariance matrix. $\mathcal{C}$ can be fixed or adapted along the iterations.

From the initial point we have a new candidate according to a multivariate normal distribution; if the candidate is accepted, we we will then restart evaluation from it. We are collecting some information allowing us to shape the target distribution.

7.5.5.1 Algorithm RW-MCMC (C known)

Initialize a matrix D_θ and a vector V_L
Randomly generate θ_1
Compute $L(\theta_1) = L_1$
For $i = 1 : N$
sample $Z \sim N(D, \mathcal{C})$
compute $\theta_2 = \theta_1 + Z$
compute $L_2 = L(\theta_2)$
compute ratio $= \frac{L_1}{L_2}$
sample $u \sim \mathcal{U}(0, 1)$
IF $u <$ ratio
$L_1 = L_2$
$\theta_1 = \theta_2$
END
IF $i >$ warm-up
$D_\theta = [D_\theta; \theta_1]$
$V_L = [V_L; L_1]$
END
END

We want to find a good accept-reject tradeoff to reach a satisfactory convergence.

Pros

- the object function is evaluated once per iteration
- the target distribution can be built form the samples
- we only need the last point to restart
- we can collect info on the model: variability on the output + sensitivity on parameters
- random selection helps us to avoid local minima

Cons

- convergence may be slow

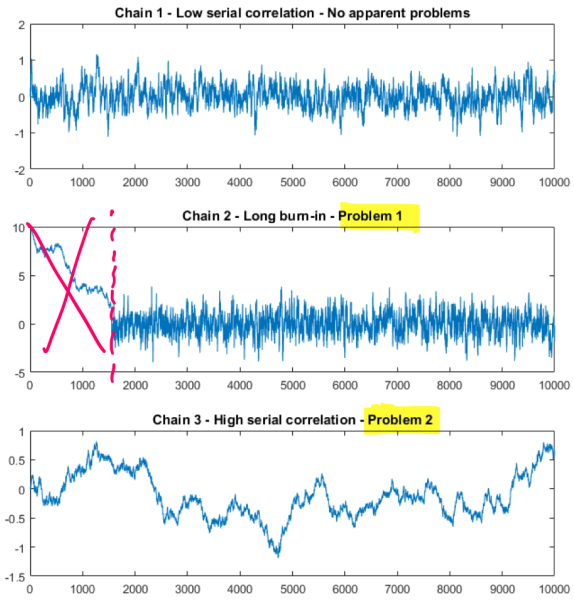

Figure 7.11: Diagnostic 1

- sampling distribution may affect the results
- each MCML cannot be parallelized
- burn-time
- diagnostics are heuristics

7.5.5.2 Diagnostics

We can exploit diagnostics to check if the parameters are good and understand whether we are done or not.

1. chain 1: expected MCMC plot
2. chain 2: we require a longer burn-in, we have not reached the target distribution
3. chain 3: we do not have enough information, the number of iterations is too small

The desired output of MCMC should look like the following plots:
Another more analytical approach is **sample split**. If we split the plot in different regions, do they have the same mean? Do we observe consistency? In chain 2 we need longer warm up, in 3 we need more samples.
Differently from gradient methods, here things are a bit more hard to interpret. We know that eventually we will reach oscillations around the global optimum, but we have no guarantee.

Diagnostics are based on the observation of the results, so they might be biased by our belief and they do not provide an easy way to read "certificate" of convergence.

Another approach may be run more MCMC in parallel and see if they all converge to the same distribution. However, it is again not a certificate of convergence to the global optimum.

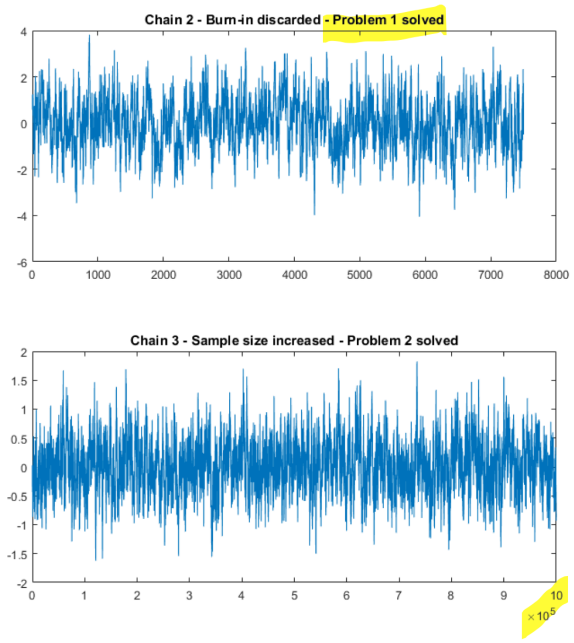

Figure 7.12: Diagnostic 2

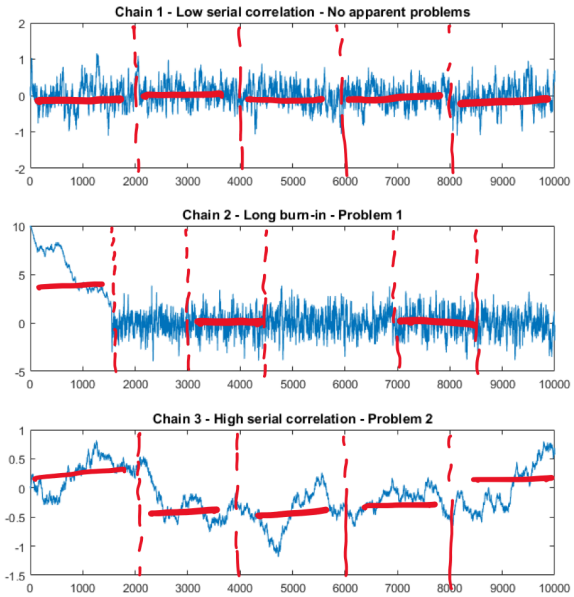

Figure 7.13: Diagnostic 3

Last observation: constraints and bounds can be included easily in the proposal distribution or in the likelihood.

7.6 Heuristics and the genetic algorithm

"Heuristics is a fancy name for trial-and-error".

Heuristics mimic natural selection i.e. follow nature inspired procedures. At the growing of computational power, they become feasible ways to solve optimization problems. There are no warranties on the exactness of the solutions, but often times the results are of high quality. In addition, heuristic algorithms are easy to implement, are general and can include *constraints* (even complex ones).

Example: timetable for plane departures. Deciding when a plane lands is not only a function of flight time, we should take into account delays, passengers, luggages, crew, ... $\rightarrow$ constraints. In biological problems we can include relationship among variables which should be satisfied, allowing high flexibility.

There are many famous heuristic algorithms:

- simulated annealing: used in physics, match thermal dispersion
- ant colonies: ant are able to solve many problems e.g. supply chain
- covariance matrix adaptation evolution strategy: performs similarly to adaptive MH algorithm

All these methods have something in common: EVOLUTION STRATEGY.

For observing evolution, we require to have a **population** of candidate solutions - not only by this method, e.g. MCMC. We then select candidates according to their **fitness function** (objective function). The changes in the populations occur as results of variations on the current population.

7.6.1 The genetic algorithm

The genetic algorithm (GA) is a family of evolutionary strategies, introduced in 1975 by John Holland. It encodes tentative solutions in chromosomal like structures. Evolution occurs as reproductive opportunities for the fittest. External variation is introduced through *mutations*.

Fundamental steps

- encoding of the chromosomes
- generation of an initial population
- fitness evaluation
- parents selection
- reproduction (crossover)
- mutation
- new population

The process is repeated from the new population to fitness evaluation. We need to understand how to encode our problem and how the selection of the mutation and reproduction are performed.

7.6.1.1 Chromosome encoding

Chromosome encoding is performed through bit strings; we have a long entity θ (chromosome) divided into sections (genes).

7.6.1.2 Generation of an initial population

Analogously to the starting points of a multi-start:

- Random
- Latin hypercube
- Orthogonal sampling

7.6.1.3 Fitness evaluation

The objective function for the current population could be picked from known functions e.g. sum of squares, likelihood or more general formulations. As long as there is a connection between the fitness number and candidate selection, the function is fine. MCMC was using one candidate at a time at each iteration, while gradient methods were computing the gradient using information from integration. In this case, the size of the population will determine the number of calls.

7.6.1.4 Parent selection

The parents can be selected through:

- threshold based selection - select best k parameters
- random based selection
  - Example (similar to Gillespie, we use fitness instead of the propensity):
    From $f^1, \ldots, f^N$ compute $\sum_{i=1}^{N} f_i = f_0$
    Generate random number $j \sim \mathcal{U}(0, 1)$
    Select the smallest k such that $\sum_{i=1}^{k} f_i > j f_0$
    Clone θ^k in an intermediate population

  "Roulette selection", the area of a circle is covered by each chromosome fitness proportionally. The idea is to spin the wheel and select the chromosome where it stops.

7.6.1.5 Reproduction

From the intermediate population we randomly select two individuals u, v and a gene for cross over t. We recombine parent chromosomes and add new offspring to the new population. This procedure is only inheriting information from the previous generation, we are missing mutations.

7.6.1.6 Mutation

The offspring may or may not mutate according to a certain probability. We denote p the probability that a gene of the new offspring mutates.

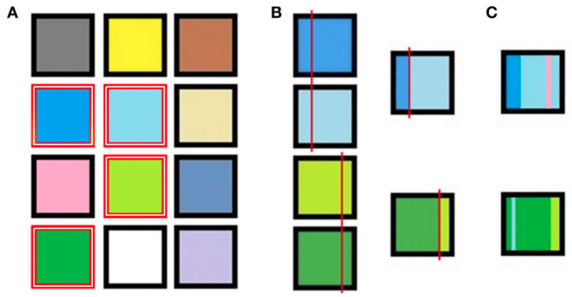

Figure 7.14: From Reali et al 2017 - GA procedure example

Input: a fitness function c that measures the goodness of the fit, for example (Equation 5); the population size N; the rate of mutation σ; the maximum number of generations G.
Output: the best candidate solution $\tilde{p}$ after G generations.

Map the parameters into strings of length l;
Generate an initial population of strings $P = \{p_1, ..., p_N\}$;
for G *times* **do**
  $P' = \emptyset$;
  Compute $f_i = c(p_i), i = 1, ..., N$ and $f_0 = \sum_{i=1}^{N} f_i$;
  for N *times* **do**
    Generate a random number $j \sim U(0, 1)$;
    Determine the smallest integer k such that $\sum_{i=1}^{k} f_i > jf_0$;
    Update $P' = p_k \cup P'$;
  end
  $P = \emptyset$;
  for N *times* **do**
    Generate two integer random numbers $m, n \sim U(1, N)$;
    Select $p_m, p_n \in P'$;
    Generate an integer random number $t \sim U(1, l)$;
    $\tilde{p} = \{p_m\{1 : t\}, p_n\{t + 1 : l\}\}$;
    for $i = 1, ..., l$ **do**
      with probability σ randomly variate $\tilde{p}\{i\}$;
    end
    Update $P = \tilde{p} \cup P$;
  end
end
Determine the best solution $\tilde{p}$ such that $c(\tilde{p}) = \min_{p \in P} c(p)$;

Algorithm 3: A simple Genetic Algorithm.

Figure 7.15: From Reali et al 2017 - GA pseudocode

7.6.2 Genetic algorithm pseudocode

Observations on specifications:

1. how many times do we go? Here we usually start by defining a number of generations
2. size of each population, use function to decide

The cost function is as general as possible. GA is sometimes used for hyperparameters tuning e.g. we can train a NN inside the algorithm. This is not feasible with Markov Chains or gradient methods.

However, we do not know whether the new generation is better than the previous one.

7.6.2.1 Cons

It is quite computationally demanding, as for N times we compute the likelihood/cost/fitness just for selecting new parents. Next, we are not directly getting direction, the reduction of the fitness function is not driven by the dimension of the population. Close to the optimum it converges slowly.

7.6.2.2 Pros

Each one of the computations can be parallelized. Some might argue that it can be used for everything. It is very general and obtains good results. f can vary sometimes; the algorithm can use an obj function without the constraints, that can be added as penalties after some generations.