University of Warsaw

Faculty of Mathematics, Informatics and Mechanics

Maja Milewska (Czoków)

Spring systems learning mechanical behaviour

PhD dissertation

Supervisor:
prof. dr hab. Jacek Miękisz
Institute of Applied Mathematics and Mechanics
Faculty of Mathematics, Informatics and Mechanics
University of Warsaw

May 2021

Author's declaration:
aware of legal responsibility I hereby declare that I have written this dissertation myself and all the contents of the dissertation have been obtained by legal means.

.....	.....
date	Author's signature

Supervisor's declaration:
the dissertation is ready to be reviewed.

.....	.....
date	Supervisor's signature

Contents

Table of Contents **3**

Abstract **7**

Streszczenie **9**

1 Introduction **11**

1.1 Motivation 11

1.2 Research methodology 12

1.3 Thesis structure 13

1.4 Main dissertation results 14

2 Spring system model **17**

2.1 Newtonian dynamics 17

2.2 Spring system model and its dynamics 19

2.2.1 Partition of the nodes 22

2.2.2 Relaxation 23

2.3 System adaptation for mechanical behaviour 25

2.3.1 Goal 25

2.3.2 Parametric learning algorithm 26

2.3.3 Relation between parametric learning and backpropagation algorithm 29

2.3.4 Determining of non-stabilised nodes 30
2.4 Generating graph $\mathcal{G}$ topology for adaptive spring systems . . . 30
2.5 Stability of equilibrium states 32
2.5.1 Learning with a noise factor 33
2.5.2 Exploration of the Hamiltonian profile 34
2.6 Conclusions 36

3 Protein model **37**
3.1 Introduction to biology of proteins 38
3.2 Implementation of the spring system method 40
3.3 Results 42
3.3.1 Effectiveness of the approach 43
3.3.2 Physical properties of protein like systems 46
3.3.3 Graphical representation of trained protein 49
3.4 Conclusions 52

4 Numerical results **53**
4.1 Prerequisites 54
4.2 Construction of a synthetic learning problem 54
4.2.1 Generation of a set of training examples 55
4.2.2 Setting initial values of spring parameters 56
4.3 Relaxation procedure 56
4.4 Properties of parametric learning algorithm 58
4.4.1 Dynamics of error Φ during adaptation process 59
4.4.2 Stop condition 62
4.5 Number of resources of a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ 64
4.5.1 Explored cases 64
4.5.2 Conclusions 70
4.6 Energy profile and noise factor properties 72
4.7 Conclusions 76

5 Conclusions **77**
5.1 Summary . . . 77
5.2 Further research . . . 78

A Rigid graphs **79**
A.1 Rigid graph . . . 79
A.2 Generic rigidity . . . 81
A.3 Henneberg constructions . . . 83
A.4 Degrees of freedom of rigid graph . . . 84

B Pseudocode **87**
B.1 Relaxation . . . 87
B.2 Parametric learning algorithm . . . 90
B.3 Stop condition . . . 93
B.4 Numerical precision of the calculation of the learning algorithm 93
B.5 Algorithm generating graph topology . . . 95
B.6 Algorithm for generation of training examples . . . 97
B.7 A noise factor . . . 99
B.8 Noise robustness . . . 100

C Backpropagation algorithm **101**
C.1 Artificial neural networks . . . 101
C.2 Gradient descent algorithm . . . 103
C.3 Sigmoidal perceptron . . . 104
C.4 Backpropagation algorithm . . . 106

Acknowledgments **115**

Bibliography **117**

Abstract

Spring systems are commonly employed to model properties of real physical objects. They are frequently used to illustrate how a solid or a microscopic body distorts under the influence of external forces. The aim of this dissertation is to apply a reverse approach. Namely, in our work we ask a question how the spring system should be constructed in order to react to external forces in a predesigned manner. Specifically, the main task in our model is to find for each harmonic spring, belonging to the system, the values of the parameters (the rest length, the elastic constant) such that, after acting of external forces on the system, it distorts in a proper way (nodes considered as observed nodes are shifted in the desired way). To achieve this aim we design an algorithm, which in sequential steps alters spring parameters to teach the system behaviour defined by the set of its training examples.

Additionally, we employ the developed mathematical framework to build a spring system with topology representing a protein structure. In this case, the trained spring system moves through the reaction path of the protein, whose *initial* and *final* states are retrieved from the open data base Protein Data Bank [5]. Our method gives us a model of the protein which can be easily used in molecular simulations.

The main results of the dissertation are published in [11] and [12]. Other results concerning our model, but not described here, are published in [13].

KEYWORDS: spring systems, learning systems, protein conformations

Streszczenie

Systemy sprężynowe uczące się mechanicznych zachowań

Systemy sprężynowe są powszechnie wykorzystywane do modelowania własności rzeczywistych obiektów fizycznych. Często są używane do obrazowania jak ciało stałe albo obiekt mikroskopowy odkształci się pod wpływem sił zewnętrznych. Celem tej rozprawy jest zastosowanie odwrotnego podejścia. Mianowicie w naszej pracy zadajemy sobie pytanie jak system sprężynowy powinien być skonstruowany, żeby zareagował na zewnętrzne siły w z góry zdefiniowany sposób. W szczególności, głównym celem naszego modelu jest znalezienie dla każdej sprężyny harmonicznej należącej do systemu wartości parametrów (długości spoczynkowej, współczynnika sprężystości), takich że po zadziałaniu na nie zewnętrznych sił system odkształci się w odpowiedni sposób (wierzchołki oznaczone jako wyjście systemu przesuną się w pożądany sposób). Aby osiągnąć ten cel, zaprojektowaliśmy algorytm, który w kolejnych krokach modyfikuje parametry sprężyn, w celu nauczenia systemu zachowań zdefiniowanych przez przykłady uczące.

Dodatkowo użyliśmy nasz aparat matematyczny do budowy systemów sprężynowych o topologii reprezentującej strukturę białek. W tym przypadku, wytrenowane systemy sprężynowe poruszają się wzdłuż ścieżek aktywności białek, których stan początkowy i końcowy są pobrane z bazy Protein Data Bank [5]. Nasza aparat matematyczny dostarcza nam struktury sprężynowe,

które mogą być wykorzystane do symulacji białek na poziomie molekularnym.

Wyniki przedstawione w rozprawie zostały opublikowane w pracach [11] oraz [12]. Inne wyniki dotyczące naszego modelu sprężynowego, ale nie opisane w rozprawie, zostały opublikowane w [13].

SŁOWA KLUCZOWE: systemy sprężynowe, systemy uczące się, konformacje białek

Chapter 1

Introduction

1.1 Motivation

Spring systems are widely used for modelling properties of microscopic and macroscopic objects. They are applied to emulate for example disordered media in material sciences [32], elastic properties of physical structures [15, 20], system designs in architectural sciences [21], and in many other contexts. Moreover, the methodology of spring systems can also be regarded as a particular instance of finite element methods for partial differential equations, see again [15]. Elastic network models with spring systems are used to explore dynamics of proteins [4, 14].

In many mentioned above problems, numerically simulated spring systems help us to find an answer to a question how a given object behaves under the influence of factors coming from the environment. For example, they illustrate how a solid distorts or cracks under the influence of external forces or how positions of atoms of a microscopic system change as a result of heating or cooling [16].

The aim of our work is to apply a reverse methodology. According to our knowledge, this is a novel approach. Namely, our goal is to automatically

construct a system which behaves in a predesigned way. In other words, instead of checking how a given manually created spring system distorts, our method is employed to construct from scratch a spring system with required elastic properties.

Spring structures are described by rigid graphs [6, 7, 29], usually embedded in a three-dimensional space. Our approach is strongly related to automated design problems because our spring systems are subject to the usual rules of real-world physics. Additionally, the output of our mathematical model can always be directly transformed into a hardware making precisely the same tasks as computer systems.

We apply our model to find spring systems with structures and dynamics correlated with proteins. We compare physical properties of these biological objects and their artificial counterparts found by our method. In particular, dynamics of harmonic springs approximate interactions between pairs of particles. Our approach refers to the elastic network model (ENM), which is commonly used in bioinformatics (see again [4, 14]). In both models, the motion of nodes is defined by harmonic springs. In ENM all particles moves with the harmonic motion. In our method this motion is over-damped, and additionally the positions of some nodes are externally controlled. In ENM two nodes are connected if distance between them is less than the value of a model parameter. In our method any pair of nodes can be connected. It is only required to represent spring systems by rigid graphs.

1.2 Research methodology

In our dissertation, we adopt algorithmical and simulational methodologies. In order to fully validate our models, numerical simulations and analyses of data were performed. The most important algorithmical parts of our mathematical model are based on the gradient descent scheme. It is applied to

compute mechanical reactions of a given system to external forces. We also take advantage of this numerical procedure to modify parameters of the system to obtain predesigned mechanical behaviour, defined by the set of its training examples. Application of the gradient descent algorithm exhibits many features common with classical methods of supervised learning of artificial neural networks (e.g. backpropagation).

The simulation software is implemented in C++ language. To accomplish the objectives of the dissertation, it was necessary to make multiple tests for various input data. Therefore, we performed our simulations on a machine with a high computing power. In order to analyse numerical results and render plots, R package was used. Graphs and neural networks in a two-dimension space were drawn in Inkscape graphics software. Graphs in a three-dimensional space were rendered in POV-Ray software.

1.3 Thesis structure

The dissertation is organised as follows.

In Chapter 2, we provide a physical theory which is the background of the spring system model. Next, we define a problem of finding a structure (values of its rest lengths and elastic constants) with given mechanical behaviour, and we build a training algorithm solving it. We also propose a stochastic algorithm generating topology of a spring structure, which can be used to test the learning ability of the spring networks. The chapter is complemented by a description of the tools whose functions are to preserve and examine the resilience of adapted systems to noisy external stimuli.

In Chapter 3, we apply our mathematical framework described in this dissertation to create elastic structures which mimic mechanical behaviour of proteins. We evaluate the obtained results. Finally, we prove that our method is able to find structures with physical properties of real biological

objects only getting as an input their three-dimensional configurations.

In Chapter 4, a numerical analysis of the training algorithm is discussed. We prove justness of applied methods and select factors, which determine quality of the found solutions.

Finally, we summarise our results in Chapter 5, and we point out potential aims of the further research.

Technical remarks about rigid graph theory, pseudocodes of applied algorithms, and backpropagation algorithm are outlined in A, B, C Appendix sections, respectively.

1.4 Main dissertation results

The main results of this dissertation is a parametric algorithm learning mechanical behaviour (see Subsection 2.3.2). This algorithm for a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ embedded in a 3-dimension space finds parameters $k[e]$ (elastic constants) and $\ell_0[e]$ (equilibrium lengths) $e \in \mathcal{E}$, so that after pushing control nodes into predefined positions, observed nodes in the equilibrium state move as close as possible to the desired control locations. To find parameters, for which the structure $\mathcal{G}$ maps mechanical behaviour, we define function $\Phi^{(i)}$ punishing for distance between the obtained and target positions of observed nodes (see the equation 2.16 for $\Phi^{(i)}$ exact formula). Next, until $\Phi^{(i)}$ stops to decrease, we modify parameters $k[e]$ and $\ell_0[e]$ according to the following gradient descent step:

1. set $k[e] = k[e] - \rho\frac{\partial\Phi^{(i)}}{\partial k[e]}$, $e \in \mathcal{E}$, where ρ is a small learning constant,
2. set $\ell_0[e] = \ell_0[e] - \rho\frac{\partial\Phi^{(i)}}{\partial \ell_0[e]}$, $e \in \mathcal{E}$.

We can define any number of desired reactions of observed nodes on dislocations of control nodes. Each of them is called a *training example*. Then, the gradient descent step is made cyclically on subsequent examples many

times. In Section 4.5 we show how the number of training examples affects the quality of the spring system adaptation.

The remaining results of this dissertation are two algorithms described for the spring system model (see again 2) and an exemplary application of our methodology for proteins (see Chapter 3):

- The relaxation algorithm which finds energetically stable configurations of spring systems (see Subsection 2.2.2).
- The stochastic algorithm generating graph topology for given mechanical behaviour (see Section 2.4).
- The parametric learning and relaxation algorithms were applied to map real-world movements of proteins onto spring systems (see Chapter 3). Their applications to real-world protein problems show flexibility of our spring system model in various areas of science.

Chapter 2

Main aspects of a spring system model

We begin with a definition of a *spring system model* and a description of its dynamics. Physical basis for this aspects is the theory of *Newton law of motion*, which is presented in parallel. Next we specify a problem of desired mechanical behaviour ascribed to a spring system. Space of solutions for this question is explored by a *parametric learning* algorithm, which is the most vital component of this dissertation. Then we present an algorithm which can be used to build a structure for a spring system. Finally, we show how to introduce a noise to the algorithm to make its output more resilient to external factors.

2.1 Newtonian dynamics

In this section, we provide basic notions present in newtonian dynamics, in particular a harmonic (damped) motion of one particle.

Let us assume that we are given particles u and v connected by a harmonic spring. Furthermore, the position of u is kept fixed in the origin of the

Euclidean space. Then, $x(t)$ for v is equal to the length of the spring in time t, and x_0 is equal to the distance to which the length of the spring is brought back by the restoring forces interacting between u and v.

Let F denote the *force* acting upon the particle. The second law of dynamics states that the acceleration of a body is proportional to the force F acting upon it:

$$\frac{d^2x(t)}{dt^2} = \frac{F}{m}, \tag{2.1}$$

where m denotes the mass of the object. For simplification, we assume that m is equal to 1. In our model the force is the sum of two forces:

$$F = F_{restoring} + F_{friction}. \tag{2.2}$$

The force $F_{restoring}$ depends linearly on displacement $x(t) - x_0$, where x_0 is a point for which $F_{restoring}$ is equal to 0. So:

$$F_{restoring} = -h \cdot (x(t) - x_0), \tag{2.3}$$

where h is a positive constant.

The second component of F is the *friction force* $F_{friction}$, which is always against the direction of the motion and is proportional to the velocity of the particle:

$$F_{friction} = -\mu \cdot \frac{dx(t)}{dt}, \tag{2.4}$$

where μ is a positive constant.

Substituting equations 2.3 and 2.4 to 2.2 we get:

$$F = -h \cdot (x(t) - x_0) - \mu \cdot \frac{dx(t)}{dt}. \tag{2.5}$$

Next, substituting the obtained equation to 2.1 we get the following equation of the motion:

$$\frac{d^2x(t)}{dt^2} = -h \cdot (x(t) - x_0) - \mu \cdot \frac{dx(t)}{dt}. \tag{2.6}$$

In this consideration, a very high friction is assumed, what is equivalent to a large value of the constant μ. After dividing both sides of the equation by μ we obtain:

$$\frac{1}{\mu} \cdot \frac{d^2x(t)}{dt^2} = -\frac{h}{\mu} \cdot (x(t) - x_0) - \frac{dx(t)}{dt}. \tag{2.7}$$

For a large value of μ the left side of the equation is small, and in our approximation we set it to 0. Simultaneously, we assume that h is large enough to keep the restoring force non-zero. Now, the form of the equation of the motion can be presented as *over-damped harmonic motion*:

$$\frac{dx(t)}{dt} = -k \cdot (x(t) - x_0), \tag{2.8}$$

where k is equal to $\frac{h}{\mu}$. Potential energy of the particle in the position x is defined by *Hamiltonian*:

$$\mathcal{H}(x) = \frac{1}{2} \cdot k \cdot (x(t) - x_0)^2. \tag{2.9}$$

Equations 2.8 and 2.9 can be written as follows:

$$\frac{dx(t)}{dt} = -\frac{d\mathcal{H}(x)}{dx}. \tag{2.10}$$

This equation defines gradient descent dynamics and it moves the particle to the local equilibrium $x(t) = x_0$ (in this case also global equilibrium).

2.2 Spring system model and its dynamics

In this section, we present a theoretical model of a *spring system* [11]. Mechanical behaviour of spring systems is derived from dynamics of a set of interacting particles. We assume that there are many particles, embedded in a 3-dimensional space and interacting each other by spring forces. Then, we represent this system of the particles by the nodes $\mathcal{V}$ of the graph $\mathcal{G}$ and interactions between them by the edges $\mathcal{V}$ of the graph $\mathcal{G}$. Whenever two nodes u and v are connected by an edge $e \in \mathcal{E}$, we write $u \sim v$.

Definition 2.2.1. *A **spring system** is an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with a finite vertex/node set $\mathcal{V}$ and an edge/spring set $\mathcal{E} \subset \mathcal{V} \times \mathcal{V}$. The system is embedded in a 3-dimensional space, thus each vertex is a point in the Euclidean space $v \in \mathbb{R}^3$. The model can be extended to any d-dimensional space.*

The edge e has an *actual length* which is the Euclidean distance between vertices u and v and is denoted by $\ell[e]$. Furthermore, with each edge $e = \{u, v\} \in \mathcal{E},\ u, v \in \mathcal{V}$ we associate two parameters: an *equilibrium (rest) length* $\ell_0[e]$ and a *spring constant* $k[e]$.

The *spring constant* $k[e] \in \mathbb{R}_+$ is the parameter determining the spring e elastic properties.

The *equilibrium (rest) length* $\ell_0[e] \in \mathbb{R}_+$ is the parameter determining the distance, to which the length $\ell[e]$ is brought back by the restoring forces interacting between u and v ($e = \{u, v\}$). The restoring force of e acting on the node v (analogously on the node u) is equivalent to 2.8. If $\ell[e]$ is not equal to $\ell_0[e]$, we say that the spring is *perturbed away* from the equilibrium length.

We reformulate 2.8 and we obtain a net force of all springs attached to each node $v \in \mathcal{V}$. The sum of all elastic forces acting upon the node v is:

$$F_v = \sum_{e=u\sim v} k[e](\ell[e] - \ell_0[e])\frac{\bar{x}_v - \bar{x}_u}{\ell[e]}. \tag{2.11}$$

By a *configuration* $\bar{x}_\mathcal{V} = ((\bar{x}_v)_{v\in\mathcal{V}}) \in \mathbb{R}^{3\cdot|\mathcal{V}|}$ of a set $\mathcal{V}$ we call a vector of locations of all nodes $v \in \mathcal{V}$.

The *Hamiltonian function* of a whole spring system for a configuration $\bar{x}_\mathcal{V}$ is given by the formula:

$$\mathcal{H}(\bar{x}_\mathcal{V}) = \frac{1}{2}\sum_{e\in\mathcal{E}} k[e](\ell[e] - \ell_0[e])^2 \ . \tag{2.12}$$

Such definition of the energy function of a body system is used in [6] and [7]. In this dissertation, we focus on a locally asymptotically stable equilibrium of

the system, that is on a configuration to which the system is pushed by 2.12, after being slightly disturbed from it. Later, in this paragraph we describe how to provide a spring system whose all local equilibria are asymptotically stable. To determine a local minimum we let the spring system evolve in time according to the standard gradient descent dynamics as it is defined by the following equation with the initial condition:

$$\begin{cases} \frac{d}{dt}\bar{x}_{\mathcal{V}} = -\nabla\mathcal{H}(\bar{x}_{\mathcal{V}}) \\ \\ \bar{x}^0_{\mathcal{V}} = \bar{x}_{\mathcal{V}}(t_0) \end{cases} . \tag{2.13}$$

By $G[\bar{x}^0_{\mathcal{V}}]$ we denote the equilibrium configuration reached by our gradient descent evolution initiated at $\bar{x}^0_{\mathcal{V}}$, and the process of approaching $G[\bar{x}^0_{\mathcal{V}}]$ is called *relaxation*.

We want the configuration $G[\bar{x}^0_{\mathcal{V}}]$ to be locally asymptotically stable, in order to remove erratic movements of the system during numeric simulations of its relaxation. We distinguish two reasons which cause the lack of that asymptotic stability of equilibria, and we introduce special constraints in order to remove this setback.

First, the lack of the asymptotic stability of equilibria is caused by the fact that the Hamiltonian (2.12) is isometry-invariant. So, the local minima $G[\bar{x}^0_{\mathcal{V}}]$ also persist for isometric transformations of the spring system (rotations and translations of all nodes simultaneously), see Appendix A.4. In the sequel, this property is removed by introducing special constraints on configurations. Namely, we shall often *freeze* positions of certain collections of nodes in $\mathcal{V}$. Some of those *frozen* nodes are interpreted as control nodes with positions set by external intervention, such as user interaction. Other nodes will be *immobilised* to reduce the number of degrees of freedom enjoyed by the system. The manner of partition of nodes $\mathcal{V}$ and influence of this division on the system dynamics, is thoroughly discussed in next subsection.

Second, the set of continuous lengths preserving graph motions can be

larger than the set of isometric motions. By definition this is equivalent to having the graph $\mathcal{G}$ *non-rigid*. In order to rule out drawbacks arising from this circumstance, we explicitly require $\mathcal{G}$ to be *rigid*. We say that a graph is *rigid* if it is impossible to change distances between two nodes in a continuous way, without modification of the lengths of the edges, see Appendix A.

2.2.1 Partition of the nodes

In our model all nodes $\mathcal{V}$ are divided into *frozen* and *movable* ones. Positions of *frozen* nodes $\mathcal{V}_{\text{frozen}}$ are considered to be constraints of the Hamiltonian function (2.12), and positions of *movable* nodes are variables for which a minimum of the energy is sought. In terms of our basic dynamics (2.13), the introduced partition of vertices, simply means taking the derivative with respect to movable nodes only. In terms of 2.11 we only move vertices which are not declared frozen. The set $\mathcal{V}_{\text{frozen}}$ is divided into subsets:

1. A set $\mathcal{V}_{\text{con}}$ of *control nodes*. These are nodes whose positions are determined by external intervention. As a result of their dislocation, dynamics (2.13) changes positions of movable nodes.

2. A set $\mathcal{V}_{\text{fixed}}$ of *immobilised nodes*, whose positions are kept fixed in the course of the evolution of the system.

The set $\mathcal{V}_{\text{movable}}$ is divided into subsets:

1. A set $\mathcal{V}_{\text{obs}}$ of *observed nodes*, which after movement of the control nodes should be dragged by the dynamics into desirable positions. Their positions can be read by an external mechanism.

2. A set $\mathcal{V}_{*}$ of *auxiliary nodes*. These nodes ensure that the graph has a structure sufficiently rich in order to efficiently solve problems to which it is applied.

Since only the movable nodes follow the dynamics (2.13), it allows us to rewrite the notation $G[\bar{x}^0_{\mathcal{V}}]$ to the form $G[\bar{x}^0_{\mathcal{V}_{\text{movable}}}; \bar{x}_{\mathcal{V}_{\text{frozen}}}]$. The obtained form denotes the equilibrium configuration of movable nodes reached by the gradient descent evolution initiated at $\bar{x}^0_{\mathcal{V}_{\text{movable}}}$ for the given positions of frozen nodes. Furthermore, because immobilised nodes $\mathcal{V}_{\text{fixed}}$ are always in the same positions and only control nodes $\mathcal{V}_{\text{con}}$ can be dislocated by external forces, it is convenient for us to hide locations of immobilised nodes and distinguish locations of control nodes in the notation G. Eventually, we use annotation $G[\bar{x}^0_{\mathcal{V}_{\text{movable}}}; \bar{x}_{\mathcal{V}_{\text{con}}}]$ instead of $G[\bar{x}^0_{\mathcal{V}_{\text{movable}}}; \bar{x}_{\mathcal{V}_{\text{frozen}}}]$ as $\mathcal{V}_{\text{fixed}}$ remains exactly the same.

We require that the set of frozen nodes $\mathcal{V}_{\text{frozen}}$ is rich enough to reduce the number of degrees of freedom enjoyed by the rigid system during the process of its relaxation and uniquely determines the local equilibrium $G[\bar{x}^0_{\mathcal{V}_{\text{movable}}}; \bar{x}_{\mathcal{V}_{\text{con}}}]$. In other words, we want to prevent the system from isometric transformations i.e. rotations, translations or combinations both of them. In order to obtain that, for spring system in a three-dimensional space, at least 3 noncollinear nodes are required to be frozen ($|\mathcal{V}_{\text{frozen}}| \geq 3$), see Appendix section A.4. Fortunately, the set of randomly selected nodes that violates this condition has a Lebesgue measure equal to 0.

2.2.2 Relaxation

Having discussed the mathematical theory, we can now rephrase it as a spring relaxation algorithm.

Input of this procedure is defined as the graph $\mathcal{G}$ and constraints for positions of control nodes $\bar{x}_{\mathcal{V}_{\text{con}}}$.

Output of the algorithm is defined as a stable equilibrium $G[\bar{x}^0_{\mathcal{V}_{\text{movable}}}; \bar{x}_{\mathcal{V}_{\text{con}}}]$, to which the system is forced by the dynamics as described in 2.12.

We say that a node is in *an equilibrium state* or *stabilised* if the absolute

value of each coordinate of the net force acting on it is smaller than the value of a predefined positive small constant. In fact, the equilibrium state is only approximated since even in the real-world dynamics the relaxation process takes infinite time.

The **algorithm** of the relaxation procedure goes as follows:

- We stabilise perturbed nodes in a sequential way. We distinguish three types of disturbances which push nodes away from their equilibrium states (they are described in 2.3.4). As a result of each of them, proper movable nodes $v \in \mathcal{V}_{\text{movable}}$ are flagged as non-stabilised and pushed into First In First Out queue. Next, the nodes are popped out from the queue and are dislocated as it is described in the point below.

- Initially, for a given node v we calculate the net force F_v acting on it. Next, v is moved in the direction of F_v multiplied by a small positive constant $step\_size$:

$$\bar{x}_v = \bar{x}_v + step\_size \cdot F_v. \tag{2.14}$$

  We repeat these two steps for the node until the node v is in the local equilibrium state or it has been dislocated maximal, acceptable times in a row. If the second condition is satisfied and the first one not, then the node is still considered as non-stabilised and again pushed into the queue.

- If, in the previous step, the node was dislocated more than once in a row, then all its stabilised neighbours are flagged as non-stabilised (pushed into the queue). So, the previous step for each vertex is usually made many times.

- Continue the restabilisation until we obtain the local equilibrium of the system (the queue is empty).

The numerical analysis of the relaxation procedure is given in Subsection 4.3. The pseudocode of the procedure and its technical details are described in Appendix section B.1.

2.3 System adaptation for mechanical behaviour

Having introduced the spring system model and its dynamics in the context of various types of vertices, we can define our main objective (see 2.3.1).

2.3.1 Goal

The principal task of this dissertation is to find physical properties of springs $\mathcal{E}$ for a rigid graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, such that $\mathcal{G}$ has desired mechanical behaviour. More precisely, for a given in advance displacement of control nodes of the graph, the spring system dynamics moves observed nodes in the desired positions. Such mechanical dependencies are predefined. The searched solution (physical properties of springs $\mathcal{E}$) must support all the control displacements. **Input** for the stated problem is given by a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ along with a set $(E^{(i)})_{i=1}^{N}$ of *training examples*. One training example defines one snapshot of the desired behaviour of the spring system. Each training example $E^{(i)} = (\bar{y}^{(i)}_{\mathcal{V}_{\text{con}}}, \bar{y}^{(i)}_{\mathcal{V}_{\text{obs}}})$ consists of:

1. *control part* $\bar{y}^{(i)}_{\mathcal{V}_{\text{con}}}$, which specifies locations of control nodes,

2. *observed part* $\bar{y}^{(i)}_{\mathcal{V}_{\text{obs}}}$, which specifies locations of observed nodes.

Output of the problem consists of parameters $k[e]$ and $\ell_0[e]$, $e \in \mathcal{E}$ and equilibria $G[\bar{x}^0_{\mathcal{V}_{\text{movable}}}; \bar{y}^{(i)}_{\mathcal{V}_{\text{con}}}]$ for $i \in \{1, \ldots, N\}$ such that positions of observed vertices in the returned equilibria are as close as possible to the locations $\bar{y}^{(i)}_{\mathcal{V}_{\text{obs}}}$.

We define the *mean squared error function*:

$$\Phi = \Phi[(k[e], \ell_0[e])_{e\in\mathcal{E}}] = \frac{1}{N}\sum_{i=1}^{N}\Phi^{(i)}, \tag{2.15}$$

where:

$$\Phi^{(i)} = \Phi^{(i)}[(k[e], \ell_0[e])_{e\in\mathcal{E}}]$$

$$\Phi^{(i)} = \frac{1}{|\mathcal{V}_{\text{obs}}|}\sum_{v\in\mathcal{V}_{\text{obs}}}(\text{dist}(\bar{y}_v^{(i)}, G[\bar{x}^0_{\mathcal{V}_{\text{movable}}}; \bar{y}^{(i)}_{\mathcal{V}_{\text{con}}}]_v))^2. \tag{2.16}$$

Now, our objective is to modify values of parameters $k[e]$ and $\ell_0[e]$, so that the value of the error function Φ is minimised. The adaptation of the spring parameters is made by a parametric learning algorithm.

2.3.2 Parametric learning algorithm

We achieve the goal by applying a parametric learning algorithm, which is the main results of the dissertation.
The algorithm, designed by us, teaches the spring system to mimic defined in advance mechanical behaviour by making updates of local parameters in two nested loops.

1. The external loop, which is repeated until the error Φ reaches a satisfactory value or it stops to improve. Each such run is called an *epoch*.

2. The internal loop during which, sequentially, for each training example $E^{(i)},\ i \in \{1, \ldots, N\}$ a gradient descent step is made in order to minimise $\Phi^{(i)}$ (see 2.16). In this loop, we call the subprocedure described below.

Input of a subprocedure which adapts spring parameters to a given training example $E^{(i)}$ is defined by the graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and the training example

$E^{(i)}$. At the beginning of the first run of this subprocedure, the configuration $\bar{x}_{\mathcal{V}}$ is determined by data passed to the algorithm. At the beginning of each remaining run, nodes have locations determined by an equilibrium configuration obtained at the end of the previous run of the subprocedure.
Output is a set of modified parameters $k[e]$ and $\ell_0[e]$ for $e \in \mathcal{E}$, and equilibria $G[\bar{x}^0_{\mathcal{V}_{\text{movable}}}; \bar{y}^{(i)}_{\mathcal{V}_{\text{con}}}]$ minimising $\Phi^{(i)}$ for $i \in \{1, \ldots, N\}$.
Subprocedure which adapts spring parameters to the training example $E^{(i)}$ is as follows:

1. **Pushing control nodes of the network to their positions in the i^{th} training example.**

   Set the positions of control nodes as follows $\bar{x}_{\mathcal{V}_{\text{con}}} = \bar{y}^{(i)}_{\mathcal{V}_{\text{con}}}$. The new locations of control nodes are constant until the end of this subprocedure, which adapts the system to the i^{th} training example. Since we dislocated control nodes, the system is not in an equilibrium point. So, we let the system evolve to an equilibrium state $G[\bar{x}^0_{\mathcal{V}_{\text{movable}}}; \bar{x}_{\mathcal{V}_{\text{con}}}]$, where $\bar{x}^0_{\mathcal{V}_{\text{movable}}}$ are current positions of movable nodes.

2. **Saving the current configuration of the network.**

   The configuration $G[\bar{x}^0_{\mathcal{V}_{\text{movable}}}; \bar{x}_{\mathcal{V}_{\text{con}}}]$, got in the previous step, and current values of parameters $p \in \bigcup_{e\in\mathcal{E}}\{k[e] \cup \ell_0[e]\}$, are initial conditions for the remaining steps of this subprocedure. For the needs of the further calculations, we save the value of the error function $\Phi^{(i)}$ (see 2.16) for the local minimum $G[\bar{x}^0_{\mathcal{V}_{\text{movable}}}; \bar{x}_{\mathcal{V}_{\text{con}}}]$.

3. **Estimation of partial derivatives of function $\Phi^{(i)}$ with respect to parameters $p \in \bigcup_{e\in\mathcal{E}}\{k[e] \cup \ell_0[e]\}$.**

   Next, we make iteration throughout the parameters $p \in \bigcup_{e\in\mathcal{E}}\{k[e] \cup \ell_0[e]\}$, in order to estimate partial derivatives of function $\Phi^{(i)}$ with

respect to p ($\frac{\partial \Phi^{(i)}}{\partial k[e]}$ and $\frac{\partial \Phi^{(i)}}{\partial \ell_0[e]}, e \in \mathcal{E}$). Each iteration we start with the constraints as stored in the previous step.

The partial derivative of function $\Phi^{(i)}$ with respect to p is approximated with application of the following formula:

$$\frac{\partial \Phi^{(i)}}{\partial p} \simeq \frac{\Phi^{(i)}[(p + \Delta p)] - \Phi^{(i)}[(p)]}{\Delta p}. \tag{2.17}$$

Notice that the value of $\Phi^{(i)}[(p)]$ is equal to the value $\Phi^{(i)}$ saved in the previous step, and for each p, in order to calculate $\Phi^{(i)}[(p + \Delta p)]$, the relaxation procedure has to be run.

The pseudocode of the algorithm estimating partial derivatives can be found in Appendix section B.2.

4. **Parameters update about the value of their gradient.**

   It is done by moving the parameters in the direction opposite to the gradient approximation:

   (a) Set $k[e] = k[e] - \rho \cdot \frac{\partial \Phi^{(i)}}{\partial k[e]},\ e \in \mathcal{E}$,

   (b) Set $\ell_0[e] = \ell_0[e] - \rho \cdot \frac{\partial \Phi^{(i)}}{\partial \ell_0[e]},\ e \in \mathcal{E}$.

   By $\rho > 0$ we denote a *learning rate*. Since the spring parameters have changed, the relaxation procedure has to be repeated. Next, the value $\Phi^{(i)}$ is calculated again. If as a result of the gradient modification, the value of the error function decreases or slightly increases, changes are accepted, otherwise they are rejected.

   After the adaptation of the spring system to a given training example $E^{(i)}$, the values of error functions for the remaining training examples $E^{(j)},\ j \neq i$ sometimes may increase. But generally Φ has decreasing trend, as expected.

The algorithm returns spring parameters for the graph $\mathcal{G}$ and equilibria $G[\bar{x}^0_{\mathcal{V}_{\text{movable}}}; \bar{y}^{(i)}_{\mathcal{V}_{\text{con}}}]$ for $i \in \{1, \ldots, N\}$ reached during the epoch for which the spring system has the smallest value of the error function Φ. The numerical analysis of the parametric learning algorithm is given in Section 4.4. The pseudocode of the procedure and its technical details are described in Appendix section B.2.

2.3.3 Relation between parametric learning and backpropagation algorithm

The specification of the problem solved by the parametric learning algorithm is very alike to the one solved by the backpropagation algorithm. For each algorithm, we are given a set of pairs of *input* and *output* vectors (in our case we call them control and observed nodes). Pairs are called *training examples*. Each algorithm operates on a structure represented by a graph. For BPA it is called a *neural network*. The aim of each of them is to teach the graph to map input vectors onto output vectors. By *teaching*, we mean *adapting* parameters assigned to a structure.

The backpropagation algorithm iterates many times in order to adapt neural network to the training examples. During each such time step the collection of parameters is modified sequentially and only one time for each training example. Such iteration is called an *epoch*. The adaptation of parameters to a given training example is obtained by taking a *gradient descent step* C.2, which minimises values of the error function defined for the training examples. As we can see in previous subsection, the parametric learning algorithm in the same way applies the *gradient descent scheme*.

2.3.4 Determining of non-stabilised nodes

As it was presented in Subsection 2.3.2, in the parametric learning algorithm nodes of a stabilised spring system are perturbed away from an equilibrium state in 3 different cases. Below, we list these cases along with a subset of nodes flagged as disturbed from their equilibrium state.

- **Case 1.** All nodes as a result of moving all spring parameters in the direction of the gradient of the error function Φ.
- **Case 2.** Neighbours of control nodes as a result of transition of control nodes positions from one training example locations to the second one.
- **Case 3.** Two nodes connected by a spring, whose parameter is modified in order to calculate the partial derivative of the error function Φ with respect to the parameter.

2.4 Generating graph $\mathcal{G}$ topology for adaptive spring systems

Here we present a procedure which generates graph topology on the basis of the set of training examples. In previous section, we assumed that input data for the parametric learning algorithm (a set of training examples and a non-adapted spring system) are given in advance. Now, we would like to find graph topology as well (see also [12]). The pseudocode of the procedure generating simple set of training examples is described in Appendix section B.6.

Input: Control and observed positions for the first training example $E^{(1)} = (\bar{y}^{(1)}_{\mathcal{V}_{\text{con}}}, \bar{y}^{(1)}_{\mathcal{V}_{\text{obs}}})$. Constants $c_{aux}, c_{\text{fixed}}$ and $c_{edge} \geq 3$, which are the number of auxiliary nodes, number of immobilised nodes and a coefficient proportional to an average node degree in the output graph $\mathcal{G}$, respectively.

Output: Topology of a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ along with its configuration $\bar{x}_{\mathcal{V}}$ (without initial values of spring parameters). Because $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is rigid, there exists a minimally rigid subgraph $\mathcal{G}' = (\mathcal{V}, \mathcal{E}') \subseteq \mathcal{G}$. Since $\mathcal{G}'$ is minimally rigid, there holds $|\mathcal{E}'| = 3|\mathcal{V}| - 6$ (removing an edge yields losing rigidity by the graph), for more details see A.2.

Our algorithm is based on the Henneberg construction method, which is an inductive approach that creates a minimally rigid graph in a $d-$dimensional space (in our case 3). This inductive construction starts from the complete graph with the 4-vertex clique K_4 at the first step, and then it adds a new node with three edges linking it to the existing graph. For more details concerning minimal rigidity and Henneberg constructions see A.3.

At least 3 noncollinear frozen nodes $|V_{frozen}| \geq 3$ are required to prevent the system from rotating, translating or a combination of them. Furthermore, the more frozen nodes are present, the less edges are required to obtain a rigid graph. For simplification we always connect a new node to the system with at least 3 edges.

Algorithm:

1. Create a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with empty sets: $\mathcal{V}_*$, $\mathcal{V}_{\text{fixed}}$, $\mathcal{V}_{\text{con}}$, $\mathcal{V}_{\text{obs}}$ and $\mathcal{E}$.

2. Randomly pick 4 nodes uniformly distributed in the ball with the radius $r \in \mathbb{R}_+$ and centered at the point $c \in \mathbb{R}^3$. Add the random nodes to the set $\mathcal{V}_*$, link each pair of these nodes with an edge.

3. Randomly pick $c_{aux} + c_{\text{fixed}} - 4$ nodes uniformly distributed in the same ball. Sequentially add them to the set $\mathcal{V}_*$ always requiring that a new node is connected to c_{edge} nearest and already added to $\mathcal{V}_*$ vertices.

4. Fix c_{fixed} nodes (move them from $\mathcal{V}_*$ to $\mathcal{V}_{\text{fixed}}$). If two immobilised nodes are connected by a spring, remove it.

5. Sequentially add observed nodes to the set $\mathcal{V}_{\text{obs}}$. The number of these nodes is determined by the number of positions in the observed part $\bar{y}^{(1)}_{\mathcal{V}_{\text{obs}}}$. Each new node is connected to c_{edge} nearest nodes already existing in the set $\mathcal{V}$, and the respective location specified in $\bar{y}^{(1)}_{\mathcal{V}_{\text{obs}}}$ is assigned to it.

6. Sequentially add control nodes to the set $\mathcal{V}_{\text{con}}$. The number of these nodes is determined by the number of positions in the control part $\bar{y}^{(1)}_{\mathcal{V}_{\text{con}}}$ Each new node is connected to c_{edge} nearest nodes in the set $\mathcal{V}_{\text{movable}}$, and the respective location specified in $\bar{y}^{(1)}_{\mathcal{V}_{\text{con}}}$ is assigned to it. In order to eliminate the rigid motions of the graph $\mathcal{G}$, it has to be satisfied $|\mathcal{V}_{\text{fixed}}| = c_{\text{fixed}} + |\mathcal{V}_{\text{con}}| \geq 3$.

7. Return the obtained structure $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ (with its configuration $\bar{x}_{\mathcal{V}}$ and without initial values of spring parameters).

The pseudocode of the procedure and its technical details are described in Appendix section B.5.

2.5 Stability of equilibrium states

In our learning algorithm, the positions of control nodes between various control parts are changed in a single discrete step, see 2.3.2. In real-world objects, which can be implemented by spring systems, these transitions are made in a different manner. The positions of their control nodes are changed in a continuous way. Simulations of such movements are not applied in our model during the learning process since it would last much longer.

On the other hand, an adapted spring system which was trained in the discrete way can behave unexpectedly during real-world movements. In other words, such system moving from $E^{(i)}$ to $E^{(j)}$ in a continuous way ($E^{(i)}$ and $E^{(j)}$ are not necessarily sequential training examples), may trap behind a

barrier of the high energy and end up in a various equilibrium than adapted for $E^{(j)}$. This can happen because during the stage of adjustment of spring parameters there are not only found local equilibria minimising $\Phi^{(i)}$ for all training examples, but also there are automatically selected *tracks*, which are used by the system to switch between these configurations. If transitions between control nodes of the adapted system are defined in a different manner than during the learning phase, the adapted system might veer off these tracks.

To minimise these setbacks, we introduce perturbations of positions of control nodes during evolution of the parametric learning algorithm. Noise is frequently added to training procedures to increase robustness of the final system, and spring systems are not exception. The procedure of adding a noise during the learning phase is given in next Subsection 2.5.1. The algorithm evaluating this modification is described in Subsection 2.5.2. Numerical tests exploring influence of the noise factor on dynamics of trained systems are presented in Subsection 4.6.

2.5.1 Learning with a noise factor

A noise factor pushes movable nodes to locations to which they can be moved by the dynamics if control nodes are dragged in an unexpected way. The modification, if applied, is made at the beginning of each subprocedure described in 2.3.2, which adapts the spring system to a given training example $E^{(i)}$. During this modification positions of control nodes are perturbed and the spring system is relaxed. As a result, after control nodes are set as follows $\bar{x}_{\mathcal{V}_{\text{con}}} = \bar{y}^{(i)}_{\mathcal{V}_{\text{con}}}$ (this is a regular step of the parametric learning algorithm), the dynamics has to always drag movable nodes from very various positions.
Input is a graph $\mathcal{G}$ and control parts of training examples $(\bar{y}^{(i)}_{\mathcal{V}_{\text{con}}})_{i=1}^{N}$.
Output is a stable equilibrium $G[\bar{x}^0_{\mathcal{V}_{\text{movable}}}; \bar{x}'_{\mathcal{V}_{\text{con}}}]$, where the constraint $\bar{x}'_{\mathcal{V}_{\text{con}}}$

is a random combination of the control parts $(\bar{y}^{(i)}_{\mathcal{V}_{\text{con}}})_{i=1}^{N}$.
Algorithm of perturbation of control nodes positions is as follows:

1. For each control node $v \in \mathcal{V}_{\text{con}}$ we calculate point $m_v \in \mathbb{R}^3$, which is the weighted sum of all positions defined for a given node v in its control parts $(\bar{y}^{(i)}_v)_{i=1}^{N}$. The applied vector of weights for a given control node is picked randomly. It has N positive coordinates, whose values sum to 1.

2. Then, for each control node v, we randomly pick point $\bar{x}'_v$, uniformly distributed in the ball $B(m_v, noise\_radius)$, where $noise\_radius$ is the algorithm parameter. Next, we set $\bar{x}_v = \bar{x}'_v$.

3. Finally, we set $\bar{x}^0_{\mathcal{V}_{\text{movable}}} = \bar{x}_{\mathcal{V}_{\text{movable}}}$, and we let the system evolve to an equilibrium state $G[\bar{x}^0_{\mathcal{V}_{\text{movable}}}; \bar{x}'_{\mathcal{V}_{\text{con}}}]$, which we return.

This optional modification in the parametric learning scheme is made in the step 5 of the Algorithm 2 and its pseudocode is presented by the Algorithm 6 in Appendix section B.

2.5.2 Exploration of the Hamiltonian profile

Here, we describe an algorithm which can be used to evaluate how the presence or lack of a noise factor during the learning process affects a trained system. This algorithm simulates continuous changes of positions of control nodes. Since we are confined by time and computer architecture, we emulate them by taking small discrete steps which drift control nodes configuration $\bar{x}_{\mathcal{V}_{\text{con}}}$ from $\bar{y}^{(i)}_{\mathcal{V}_{\text{con}}}$, and near it to $\bar{y}^{(j)}_{\mathcal{V}_{\text{con}}}$. After each such step the relaxation process is run. Tracks of transition of control nodes can be defined by various curves.
Input of this algorithm is an adapted spring system $\mathcal{G}$, its training examples $(E^{(i)})_{i=1}^{N}$, and a set of equilibria for all $(E^{(i)})_{i=1}^{N}$ returned for the system $\mathcal{G}$

by the parametric learning algorithm (equilibria obtained by the dynamics during the training epoch with the lowest error Φ). These equilibria are denoted as $(G^{(i)}_{patern})_{i=1}^{N}$.

Output returned by the algorithm is the value of the metric Ψ, which measures distortion between the adapted equilibrium $G^{(j)}_{patern}$ and the equilibrium obtained as a result of a continuous transition from $E^{(i)}$ to $E^{(j)}$.

Definition 2.5.1. *By Ψ is denoted the average distance between positions of movable nodes in two equilibrium states reached by the dynamics for a given graph $\mathcal{G}$. Both equilibria are obtained by the relaxation procedure initiated with the same constraint on control nodes locations $\bar{x}_{\mathcal{V}_{\text{con}}}$ and with various initial conditions for locations of movable nodes: $\bar{x}'_{\mathcal{V}_{\text{movable}}}$ and $\bar{x}''_{\mathcal{V}_{\text{movable}}}$. If the value of the function is equal to 0, the compared equilibria are equal.*

$$\Psi(G[\bar{x}'_{\mathcal{V}_{\text{movable}}}; \bar{x}_{\mathcal{V}_{\text{con}}}], G[\bar{x}''_{\mathcal{V}_{\text{movable}}}; \bar{x}_{\mathcal{V}_{\text{con}}}]) =$$

$$= \frac{1}{|\mathcal{V}_{\text{movable}}|} \sum_{v \in \mathcal{V}_{\text{movable}}} \text{dist}(G[\bar{x}'_{\mathcal{V}_{\text{movable}}}; \bar{x}_{\mathcal{V}_{\text{con}}}]_v, G[\bar{x}''_{\mathcal{V}_{\text{movable}}}; \bar{x}_{\mathcal{V}_{\text{con}}}]_v). \tag{2.18}$$

Algorithm:

1. We randomly choose two various training examples $E^{(i)}$ and $E^{(j)}$. To the configuration of the system $\bar{x}_{\mathcal{V}}$ we assign $G^{(i)}_{patern}$.

2. Next, for each control vertex $v \in \mathcal{V}_{\text{con}}$ we choose a curve connecting $\bar{y}^{(i)}_v$ and $\bar{y}^{(j)}_v$. We propose two manners of defining these curves. The first one and the simplest—by a straight line. The second way—a random semicircle whose ends are attached to points $\bar{y}^{(i)}_v$ and $\bar{y}^{(j)}_v$. Curves define tracks along which nodes $v \in \mathcal{V}_{\text{con}}$ move during the transition from $E^{(i)}$ to $E^{(j)}$. The transition is made in L steps (L is a positive integer parameter) during which each control vertex $v \in \mathcal{V}_{\text{con}}$ is moved along

the proper line or arc by $\frac{1}{L}$ of the length of this path in the direction of $\bar{y}_v^{(j)}$.

3. After each such small step, the system is relaxed. When control nodes are in the positions $\bar{y}_{\mathcal{V}_{\text{con}}}^{(j)}$, we denote the current equilibrium as $G_{obtained}^{(j)}$. Finally, the algorithm returns $\Psi(G_{patern}^{(j)}, G_{obtained}^{(j)})$, which in an ideal case is equal to 0.

The pseudocode of the scheme is given in Algorithm 7 in Appendix section B.

2.6 Conclusions

In this chapter, we introduced the notion of the spring system model. We defined algorithms for training and testing them:

- The relaxation algorithm 2.2.2.
- The parametric learning algorithm 2.3.2.
- The algorithm generating graph topology 2.4.
- The algorithm adding a noise factor during the learning process 2.5.1 and the algorithm which can be used to test robustness of a spring system on the added noise 2.5.2.

Numerical tests of these methods are presented in Chapter 4.

Chapter 3

Protein model

Here we use our mathematical model discussed in Chapter 2 to construct and analyse spring systems which behave like proteins. We are given a time sequence of real-world configurations achieved by a protein during performing its biological functions. Our first goal is to train the spring system, which is described in Section 2.2, to mimic the given mechanical behaviour of the protein. To achieve the aim we represent the residues of the protein by the set of vertices of the spring system. Interactions between nearby residues of the protein are modelled by harmonic springs. The spring system is trained to move through the given sequence of configurations by our parametric learning algorithm given in Subsection 2.3.2.

Our second goal is to check if spring parameters found by the parametric learning algorithm have realistic values. For the trained relaxed system we calculate the mean value of forces dependent on distance between two nodes. We fit the obtained curve to the Lennard-Jones force. In this way we show that the strength of its intermolecular interactions is coded in its reaction path.

This chapter is organised as follows. Section 3.1 is a brief introduction to biology of proteins. We describe there distinct aspects of structures of these

molecular objects and functions provided by them. In Section 3.2, we show how we map a structure of a protein and its real-world configurations onto a graph representing a spring system. Next, having the spring system trained, in Section 3.3, we assess a quality of the obtained results. In Subsection 3.3.1, we show how close the system is pushed to its training configurations, when the positions of its control nodes are dragged through them. In Subsection 3.3.2, we assess if forces acting between nodes of the trained systems in their stable equilibria have realistic values. We discuss the results in Section 3.4.

3.1 Introduction to biology of proteins

Proteins are large macromolecules which consist of amino acid residues. They exist in all living organisms, where they serve many functions:

- as antibodies they bind to viruses and bacteria to help protect the body,
- as enzymes they carry out DNA replication and most of chemical reactions in cells,
- as messenger proteins they transmit signals to coordinate biological processes between various cells, tissues, and organs,
- as structural proteins they are responsible for stiffness of biological components.

Amino acids in proteins are aligned linearly in a sequence called a *polypeptide* (proteins can consist of more than one polypeptide but we do not consider such cases in this work). There exists 22 types of amino acids. Their order in the sequence is known as a *primary structure*. Alignment of amino acids plays a crucial role in determining shapes of proteins in the three-dimensional space. Locally, the primary structure implicates regularly repeating constructions, among which the most common are alpha helices,

beta sheets, and turns. Alignment of local shapes in a sequence is called a *secondary structure*. Three-dimensional arrangement of local constructions is known as a *tertiary structure*. Such a global state and its flexible motions define biological activities of the protein. Alternative structures of the same protein are referred to as its *conformations*. Linked chains of amino acids of proteins are created by ribosomes during translation processes. As a consequence of interactions between amino acids, such sequences fold into stable three-dimensional structures, which are called *native conformations*, they are capable of performing various biological functions. Many proteins can fold as a result of chemical reactions of their amino acids, others require help of external molecular objects.

As a result of performing their functions, proteins tertiary structures can be modified. Many proteins convert between their two various locally stable states [31]. Such alternations of tertiary structures can be a consequence of proteins interactions with other molecular objects. Enzymes, for example, act as biological catalysts. Their bind to molecules called substrates and convert them into other molecules known as products. The substrate is attached to a region of a protein called an *active pocket site*. During this process, the enzyme changes its conformation from *open* to *closed* since shape of the active pocket site looks like it is being *closed* [36]. In turn, messenger proteins generate signals mostly as a result of ligands binding.

Alternative three-dimensional configurations of proteins, observed with the use of empirical methods, are described in the Protein Data Bank (PDB), see [5] and url www.rcsb.org[1]. But servers like this one do not present transitions between given conformations, since experimental methods are insufficient to explore them. Even X-ray diffraction, which is the most powerful structural technique, provides only a single frozen picture from a conformational ensemble in terms of averaged configurations.

[1]Link access date 2021-May

Movements of tertiary structures are referred to as *conformational changes* and corresponding pathways are called *classical reaction paths*. In the past years, scientists refined numerical algorithms trying to determine intermediate conformations for two predefined, *initial* and *final*, configurations, see [25], [27], [28], [35], and [40].

Molecular dynamics (MD) is the best known theoretical technique for reproducing protein movements. It uses atomistic force-field and an explicit representation of solvent [26]. But computations of movements of large molecular objects can be extremely expensive. Such problems can be tackled by *coarse-grained models*, which use reduced representations of structures, see [22] and [37]. These models, in spite of their simplicity, deliver very often results of a surprising quality.

Elastic network model (ENM) is a specific implementation of a coarse-grained approach, see [4]. In this method, positions of amino acids in proteins are usually represented by locations of their α-carbon (C_α) atoms. A *side chain* is a group of atoms specific for each type of given amino acid. The side chain is attached to the α-carbon. Interactions between C_α atoms are modelled by harmonic springs if the distance between them is less than the value of a parameter *cutoff* for selected conformations. Forces acting between remaining amino acids are negligible (they are not connected by springs). Clearly, this is not a realistic approach, since in the microscopic world physical forces decrease with a distance. Non-cutoff models based on an exponentially decaying function are more precise, like the ones proposed in [23] and [30].

3.2 Implementation of the spring system method

To obtain goals of this chapter we map a structure of a protein onto a system of springs represented by a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, see Section 2.2. The set of

vertices $\mathcal{V}$ consists of C_α atoms of the protein. There is one simplification in comparison to the classical spring system model. Here, only control and observed nodes are used $\mathcal{V} = \mathcal{V}_{\text{con}} \cup \mathcal{V}_{\text{obs}}$. In our simulations, $|\mathcal{V}_{\text{con}}| \approx 0.2 \cdot |\mathcal{V}|$ ($|\mathcal{V}_{\text{obs}}| \approx 0.8 \cdot |\mathcal{V}|$) and nodes of $\mathcal{V}$ are split randomly into sets $\mathcal{V}_{\text{con}}$ and $\mathcal{V}_{\text{obs}}$.

As an initial conformation of a given protein we choose its biologically functional state, not bound with other molecular objects. It determines the first training example $E^{(1)}$. The final state is the closed or/and ligand bound form of the same protein. It determines the last training example $E^{(N)}$. The conformations are downloaded from the PDB database. Configurations of $E^{(1)}$ and $E^{(N)}$ are aligned (rotated and translated) to minimise squared average distance between corresponding nodes. The remaining training examples constitute a reaction path between two already given states of the protein. These intermediates are calculated by algorithm GOdMD described in [35], and whose functionality is freely available on the web server cited in the paper. Moreover, an order of the intermediates in the training set is consistent with their appearance during the transition process of the protein.

Two nodes are connected by a spring if distance between them is less than the value of a parameter called *cutoff* at the initial or final training example. In our consideration, *cutoff* $= 13$Å (e.g. in [34] $8-9$Å). Equilibrium lengths $\ell_0[e]$, $e \in \mathcal{E}$, are initiated by actual lengths of respective edges $\ell[e]$ in the last training example $E^{(N)}$.

Parameters defining elastic properties of springs $k[e]$, $e \in \mathcal{E}$, before the learning process are set up to be a constant value. Parameters k are measured in $\frac{kcal}{mol \cdot \text{Å}}$, but for simplicity we treat them as unitless.

The spring system is trained to move along the reaction path of a protein by our parametric learning algorithm given in Subsection 2.3.2. But in the protein case, we modify the error function $\Phi^{(i)}$ (2.16) to adapt the system to the i^{th} training example.

The original function $\Phi^{(i)}$ penalizes a deviation of observed nodes from

their positions in the i^{th} training conformation.

The modified error function $\Phi^{(i)}_{modified}$ additionally penalizes deviations of all equilibrium lengths $\ell_0[e]$, $e \in \mathcal{E}$, from 3.8Å[2]:

$$\Phi^{(i)}_{modified} = \Phi^{(i)} + \frac{1}{|\mathcal{E}|}\sum_{e\in\mathcal{E}}(\ell_0[e] - 3.8)^2,$$

$$\Phi^{(i)}_{modified} = \frac{1}{|\mathcal{V}_{\text{obs}}|}\sum_{v\in\mathcal{V}_{\text{obs}}}(\text{dist}(\bar{y}^{(i)}_v, G[\bar{x}^0_{\mathcal{V}_{\text{obs}}}; \bar{y}^{(i)}_{\mathcal{V}_{\text{con}}}]_v))^2 + \frac{1}{|\mathcal{E}|}\sum_{e\in\mathcal{E}}(\ell_0[e] - 3.8)^2, \tag{3.1}$$

where $G[\bar{x}^0_{\mathcal{V}_{\text{obs}}}; \bar{y}^{(i)}_{\mathcal{V}_{\text{con}}}]$ is the equilibrium state of the spring system reached upon setting control nodes $\bar{x}_{\mathcal{V}_{\text{con}}} = \bar{y}^{(i)}_{\mathcal{V}_{\text{con}}}$.

In the protein case we do not use a noise factor.

3.3 Results

The aim of this section is to evaluate properties of spring systems structured and adapted to reshape like proteins. Each trained system should model in the best way the whole transition path from the initial conformation to the final one. We tested our model on 5 proteins. For each of them we made 10 training simulations with various nodes marked as control/observed ones. In Subsection 3.3.1, we show how accurate is our model in reaching adapted conformations. We also look at comparison between results of our method and the GOdMD method [35]. In Subsection 3.3.2, for the trained systems in their stable equilibria, we plot the mean value of forces over the distance between C_α residues. We compare the obtained result to the Lennard-Jones force. In Subsection 3.3.3 we show figures depicting spring systems for trained Calmodulin Human peroxiredoxin protein.

[2] Å - angstrom is a unit of length equal to 10^{-10} m, value 3.8Å is the physical equilibrium length between C_α atoms.

3.3.1 Effectiveness of the approach

In this subsection, we show how does the parametric learning algorithm manage to learn mechanical behaviour of proteins. We examine how close trained and relaxed spring systems near to their whole reaction paths. We measure it by Φ error (see Equation 2.15). We carry out comparison between results for the final conformations (measured by $\Phi^{(N)}$, see Equation 2.16) and equivalent GOdMD [35] results for 5 proteins. GOdMD starts from the initial configuration and generates a realistic reaction path targeting to the final one. The errors between target final structures and approached ones by GOdMD are measured by a root mean square deviation error RMSD and are given in Supplementary Information for [35]. The value of RMSD is calculated over the distance of all atoms of two compared structures $\bar{x}'_{\mathcal{V}}$ and $\bar{x}''_{\mathcal{V}}$:

$$RMSD(\bar{x}'_{\mathcal{V}}, \bar{x}''_{\mathcal{V}}) = \sqrt{\frac{1}{|\mathcal{V}|}\sum_{v\in\mathcal{V}}(\mathrm{dist}(\bar{x}'_v, \bar{x}''_v))^2}. \tag{3.2}$$

To unify GOdMD error with $\Phi^{(N)}$ we transform RMSD into MSD form which we define as follows:

$$MSD(\bar{x}'_{\mathcal{V}}, \bar{x}''_{\mathcal{V}}) = \frac{1}{|\mathcal{V}|}\sum_{v\in\mathcal{V}}(\mathrm{dist}(\bar{x}'_v, \bar{x}''_v))^2. \tag{3.3}$$

There is a small difference between MSD and $\Phi^{(N)}$. The function $\Phi^{(N)}$ is calculated over the set $\mathcal{V}$ with exclusion of control nodes. This difference is justified since positions of control nodes for spring systems are externally controlled, and elements of the error function for $v \in \mathcal{V}_{\mathrm{con}}$ are always equal to 0. Results of the methods are compared in Table 3.1.

Let us emphasise that intermediate conformations generated by GOdMD to train spring systems does not improve quality of the error $\Phi^{(N)}$. A spring system adapted only to the initial and final conformations should have lower error $\Phi^{(N)}$ than the same system trained to map a transition path with in-

termediates. So, without loss of generality we can compare results of both methods for the last training example.

In the following columns are:

- protein — the name of the protein,
- initial state $E^{(1)}$— the PDB code of an initial conformation, the structure is mapped onto the first training example of the spring system,
- final state $E^{(N)}$ — the PDB code of an final conformation, the structure is mapped onto the last training example of the spring system,
- number of C_α — the number of C_α atoms (amino acids) in the protein,
- $MSD^{boundary}$ — the deviation between initial and final conformations measured by MSD,
- MSD^{GOdMD} — the deviation between the final state developed by GOdMD and the real target conformation of the reaction path, see [35], it is calculated throughout all nodes,
- $\overline{\Phi^{(N)}}$ — $\Phi^{(N)}$ averaged over results for 10 simulations for a given protein
- $\overline{\Phi}$ — Φ averaged over results for 10 simulations for a given protein
- N — the number of conformations in the whole transition path.

In Table 3.1. results of our and the GOdMD methods for the final conformation are depicted in red and green colour, respectively.

protein	initial state $E^{(1)}$	final state $E^{(N)}$	number of C_α	$MSD^{boundary}$	MSD^{GOdMD}	$\Phi^{(N)}$	Φ	N
Calmodulin Human peroxiredoxin	1cfd	1cfc	148	5.33	0.04	0.57	0.60	13
Guanylate Kinase	1ex6	1ex7	186	13.32	0.08	0.32	0.30	6
Adenylate Kinase from Aquifex Aeolicus	2rh5	2rgx	202	35.00	0.15	0.45	0.45	14
(LAO) binding protein	2lao	1lst	238	22.75	0.07	0.28	0.24	7
Oligopeptide-binding protein	1rkm	2rkm	517	9.61	0.15	0.21	0.07	13

Table 3.1: Comparison of the results of the GOdMD method and the parametric learning algorithm for 5 proteins. For the final conformations, the results of the methods are denoted in green and in red, respectively.

Our algorithm adapts spring systems with a high precision to the whole reaction paths. The GOdMD method for the last conformations gets a little better results than the parametric learning algorithm. Although we consider our results to be satisfying, since the goal of our method is not only to map conformations but also code them in spring structures.

3.3.2 Physical properties of protein like systems

In this subsection, we analyse values of forces acting between residues of adapted systems in stable configurations. Initially, before the learning process, these values are nothing like in real physical objects. For the last training example they are equal to 0 since for each spring $e \in \mathcal{E}$, to its equilibrium length $\ell_0[e]$, we assign its actual length $\ell[e]$ in the final conformation, see 3.2 for the initial conditions. After the learning process, the curve which plots the mean value of forces dependent on the spring length gets a realistic shape. Here, we compare it to the shape of the same interaction described by the Lennard-Jones theoretical model. We focus on force since it drives dynamics during the relaxation processes. Its value also combines all elements searched by the parametric learning algorithm: an elastic constant, equilibrium length and system configuration.

The classic Lennard-Jones model approximates potential between a pair of molecules or atoms. Its the most common expression is:

$$V_{u\sim v}(\ell) = 4\epsilon \left[\left(\frac{\sigma}{\ell}\right)^{2n} - \left(\frac{\sigma}{\ell}\right)^{n} \right], \tag{3.4}$$

where $\ell = \text{dist}(\bar{x}_v, \bar{x}_u)$ is the inter-atomic distance between two atoms v and u, n usually is equal to 6, σ is the distance at which potential for each pair of atoms is zero, and ϵ is the depth of the potential well. Term $\left(\frac{\sigma}{r}\right)^{2n}$ describes repulsive interactions. Term $\left(\frac{\sigma}{r}\right)^{n}$ represents attractive factors. The Lennard-Jones potential is depicted in blue in Figure 3.1.

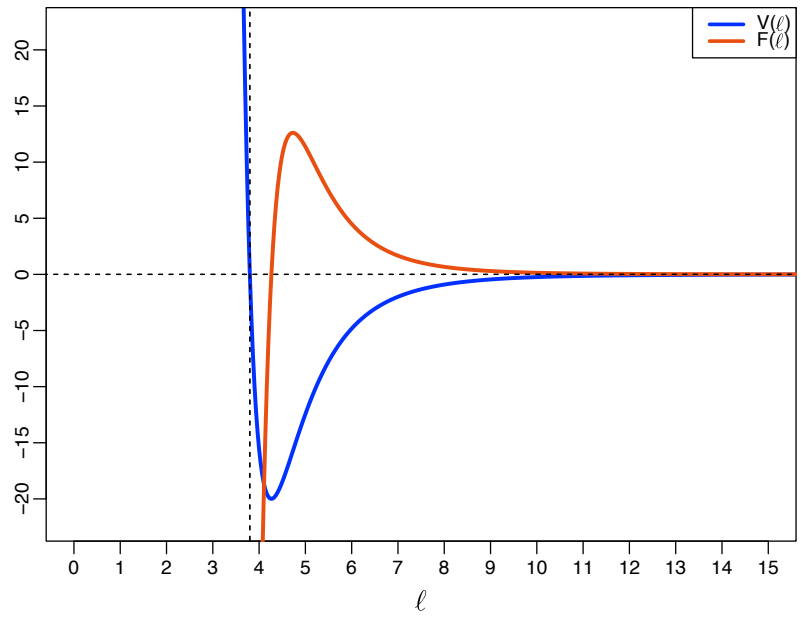


Figure 3.1: The Lennard-Jones potential and force for $\epsilon = 20$, $n = 6$, and $\sigma = 3.8$.

From the Lennard-Jones potential we derived a formula on distance dependent force:

$$LJ_F = \frac{dV_{u\sim v}(\ell)}{d\ell} = 4\epsilon \left[-2n \left(\frac{\sigma^{2n}}{\ell^{2n+1}} \right) + n \left(\frac{\sigma^n}{\ell^{n+1}} \right) \right]. \tag{3.5}$$

The plot for this formula is drawn in red in Figure 3.1.

To depict the mean value of forces dependent on distance between two nodes for adapted systems, we bucketing springs for each trained protein. This is a discretisation data method. Springs $e \in \mathcal{E}$ are grouped into bins on the basis of their actual lengths $\ell[e]$ in the final stable configuration $E^{(N)}$.

To achieve the aim we make the following steps:

- For each protein to the same bin we put springs which satisfy $\ell[e] \in [0.25 \cdot i, 0.25 \cdot (i+1))$, where $i \in \mathbb{N}$. Values $\ell' = 0.25 \cdot i$ are labels of the bins.

- Next, for each bin for a given protein p we calculate the mean value of elastic constants k and equilibrium lengths ℓ_0. We denote them by $\overline{k^p(\ell')}$ and $\overline{\ell_0^p(\ell')}$.

- We calculate the mean value of $\overline{k^p(\ell')}$ and $\overline{\ell_0^p(\ell')}$ for all training proteins, and we denote them by $\overline{k(\ell')}$ and $\overline{\ell_0(\ell')}$.

- Finally, for each bin we calculate the mean value of forces:

$$\overline{F(\ell')} = \overline{k(\ell')}(\ell' - \overline{\ell_0(\ell')}). \tag{3.6}$$

We search parameters of the Lennard-Jones force which minimises RMSD (see Equation 3.2) between corresponding points $(\ell', \overline{F(\ell')})$ and $(\ell', LJ_F(\ell'))$. The found values of the parameters are $n = 0.7$, $\sigma = 1.88$, and $\epsilon = 139.67$. RMSD for the stated problem is equal to 0.84. During solving this optimisation problem we remove outliers from the set of averaged forces $\overline{F(\ell')}$. The plot depicting $\overline{F(\ell')}$ is drawn in Figure 3.2 along with the Lennard-Jones force fitted to it. In the plot we distinguish outliers and points $\overline{F(\ell')}$ which were used to find the parameters minimising the value of the function RMSD.

The slide in interpolation for $\ell' = 13$, which is equal to the cutoff parameter, can be explained by decrement of the sample size in bins with labels $\ell' > 13$.

Matching both curves, confirms that intermolecular interactions are coded in folding paths of proteins. Of course, the Lennard-Jones force with $n = 6$ describes better real physical objects, however, taking into account simplicity of our model we find this result very satisfying.

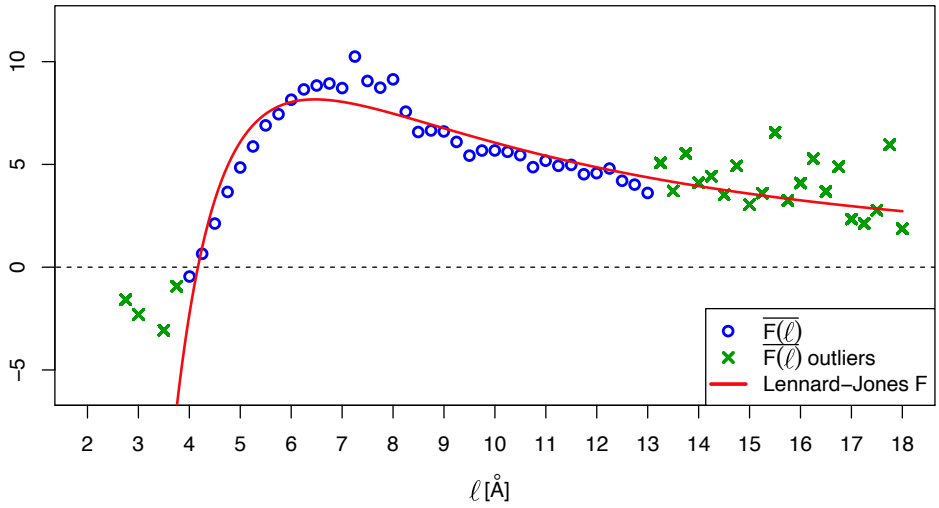


Figure 3.2: The averaged forces $\overline{F(\ell')}$ are plotted in blue and green for points used to solve the optimisation problem and outliers, respectively. In red, the Lennard-Jones force fitted to the experimental data.

3.3.3 Graphical representation of trained protein

In Figures 3.3 and 3.4 we depict trained spring systems for the initial and final conformations for Calmodulin Human peroxiredoxin protein (148 C_α residues/vertices).

Figure 3.3: An equilibrium trained for protein 1cfd, $cuttof = 11$Å, in red positions of nodes, stars represent control nodes, balls represent observed nodes and green cones indicate into their desired positions. Edges change colour from grey to black as the attracting force increases and from dark yellow to bright yellow as the repulsive force increases.

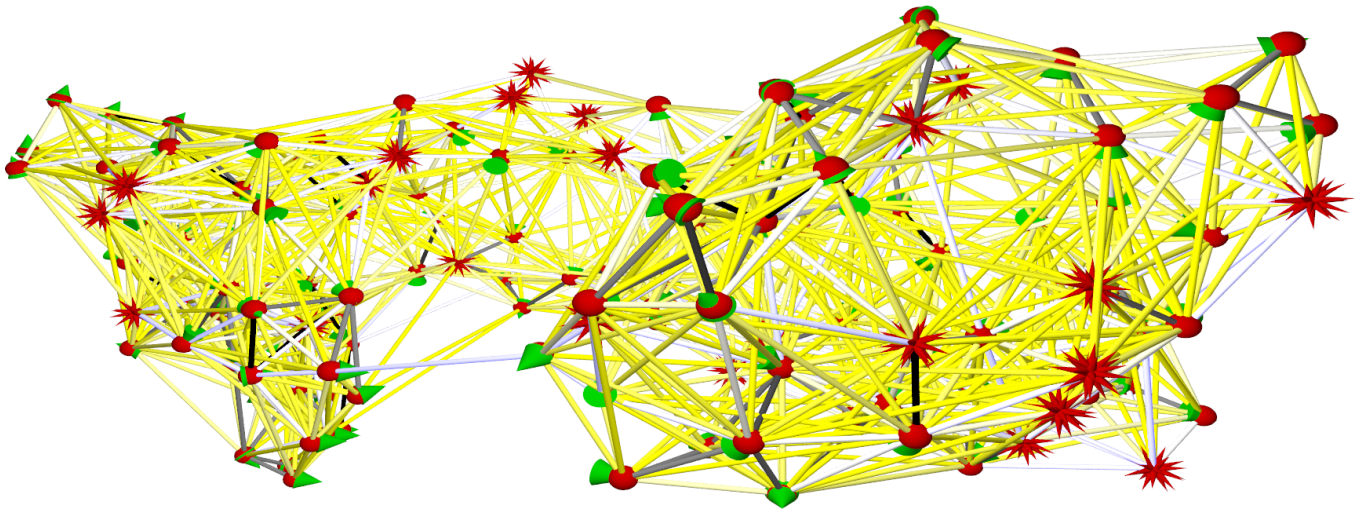


Figure 3.4: An equilibrium trained for protein 1cfc, $cuttof = 11$Å, in red positions of nodes, stars represent control nodes, balls represent observed nodes and green cones indicate into their desired positions. Edges change colour from grey to black as the attracting force increases and from dark yellow to bright yellow as the repulsive force increases.

3.4 Conclusions

In this chapter, we have proved that our method from 3-dimensional structural data is able to derive dynamics approximating behaviour of real micro-objects. In Subsection 3.3.1, we have shown that our parametric algorithm is able to store transition paths of proteins in structures of spring networks. In Subsection 3.3.2, we have plotted the mean value of forces as a function of the distance between C_α residues for trained systems in their stable equilibria. We have shown that the obtained curve fits well to the Lennard-Jones force.

Chapter 4

Numerical results

In this chapter, we explore numerical properties of the parametric learning and generating graph topology algorithms. We mainly measure how their various parameters impact on the value of the error function Φ (see 2.3.1) calculated for adapted systems. We also study time consumed by the learning process. It is not surprising that these two magnitudes are negatively correlated. We try to find the trade-off between these two features in order to get well-adapted systems in reasonable time.

In Section 4.1, we specify prerequisites for numerical tests conducted in this chapter.

In Section 4.2, we show how to automatically sample a problem, which can be an exemplary input of the parametric learning algorithm and can be applied to explore its properties.

In Section 4.3, we confirm justness of usage of the relaxation subprocedure. This scheme is extremely vital since its executions consume almost 100% of the whole learning processes.

In Section 4.4, we carry tests which confirm effectiveness of the gradient descent scheme for finding parameters of a spring system structure.

In Section 4.5, we show how the adaptation quality of a spring system,

depends on the topology of its graph $\mathcal{G}$.

In Section 4.6, we test resilience of an adapted system to local equilibria which were not developed during the learning phase of the system. It appears that the topology of the graph and the noise factor, applied during the adaptation process, are significant for stability of the system.

4.1 Prerequisites

For all numerical simulations presented in this chapter, we construct synthetic training problems in an automatic way, and next we train the returned spring systems with application of the parametric learning algorithm.

Unless stated otherwise, for each test we apply the same values of parameters of the learning algorithm and its subprocedures. If the value of a control variable is common for all test simulations, we give it in Appendix chapter B. Otherwise, it is pinpointed in respective tests prerequisites. All values of parameters and results presented in this chapter and Appendix B are unitless. Parameters ℓ and ℓ_0 are measured in magnitudes of displacement, and k in mass per a squared unit of time. For notational convenience, we omit the units.

Time consumed by simulations is measured in seconds. A processor used to make simulations was Intel(R) Xeon(R) CPU E5-2690 2.90GHz.

4.2 Construction of a synthetic learning problem

For real-word applications of our spring system model, we assume that a graph $\mathcal{G}$ and training examples are given in advance or are easy to derived. For example, for a problem of modeling of proteins, both components can be

acquired from the molecular structure of a given protein, see Section 3.2.

In this section, we put forward the process of automatic creation of the whole input data for the parametric learning algorithm, in the case they are no longer provided.

The process of automatic creation of a *synthetic problem* for the parametric learning algorithm we divide into three stages:

1. generation of a set of training examples,
2. definition of the topology of a graph $\mathcal{G}$ with its initial configuration,
3. setting initial values of spring parameters.

We have already handled the second case in Section 2.4. How to generate a set of training examples and set initial values of springs parameters we put forward respectively in 4.2.1 and 4.2.2.

4.2.1 Generation of a set of training examples

A procedure described in this subsection generates all locations of nodes given in a set of training examples $(E^{(i)})_{i=1}^{N}$, for a predefined $|\mathcal{V}_{\text{con}}|$, $|\mathcal{V}_{\text{obs}}|$, and N training examples.

Initially, the procedure randomly pick control and observed parts for the first training example from the surface of the 3d ball $B(c, r)$, where $c \in \mathbb{R}^3$ is the center of the ball and $r \in \mathbb{R}_+$ is the radius of the ball. The sphere of the ball is partitioned into two hemispheres. $\bar{y}^{(1)}_{\mathcal{V}_{\text{con}}}$ are picked from one hemisphere and $\bar{y}^{(1)}_{\mathcal{V}_{\text{obs}}}$ from the other one.

Next, for the remaining training examples $i \in \{2, \ldots, N\}$:

- randomly pick a vector $w^{(i)} \in \mathbb{R}^3$ and scale it to the length $l^{(i)} \in \mathbb{R}_+$, where $l^{(i)}$ is also a random variable, $l^{(i)} \sim \mathcal{U}(0, \theta_1)$, and $\theta_1 < r$ is a constant of the procedure,

- next, for each vertex $v \in \mathcal{V}_{con} \cup \mathcal{V}_{obs}$ randomly pick a vector $\psi \sim (\mathcal{N}(0, \theta_2^2), \mathcal{N}(0, \theta_2^2), \mathcal{N}(0, \theta_2^2))$, where $\theta_2 < r$ is a constant of the procedure and perform $\bar{y}_v^{(i)} = \bar{y}_v^{(1)} + w^{(i)} + \psi$.

The greater the constants θ_1 and θ_2 are, the harder the training problem is.

4.2.2 Setting initial values of spring parameters

Having specified the topology of the graph $\mathcal{G}$ along with its configuration we would like to set the initial values of its spring parameters.

For each $e \in \mathcal{E}$, the equilibrium length of a spring $\ell_0[e]$ is set up to be the actual length of this spring $\ell[e]$. As a result, the current configuration of the graph is in the global equilibrium state, and equilibrium lengths are determined by the initial configuration of the graph $\mathcal{G}$ constructed by the algorithm generating graph topology. Moreover, since $\bar{x}_{\mathcal{V}_{\text{con}}}$ is equal to $\bar{y}^{(1)}_{\mathcal{V}_{\text{con}}}$, and $\bar{x}_{\mathcal{V}_{\text{obs}}}$ is equal to $\bar{y}^{(1)}_{\mathcal{V}_{\text{obs}}}$, we have $\Phi^{(1)} = 0$. During the learning phase, the perfect adaptation to the first training example is destroyed, hopefully to obtain the low mean error for all training examples.

Elastic constants $k[e]$ for all springs $e \in \mathcal{E}$ initially have the same value. In our computer simulation we apply value 40. If all elastic constants initially have the same magnitude, like in our case, the initial value of the Hamiltonian function depends linearly on this value. So, we can treat this value as a scaling parameter, which has only influence on the numerical aspects of the learning algorithm.

4.3 Relaxation procedure

In this section, we measure how the profile of the Hamiltonian function 2.12 changes during the process of nodes stabilisation, see the top figure in 4.1. Simultaneously, we store the sum of values of the net forces acting on all

movable nodes:

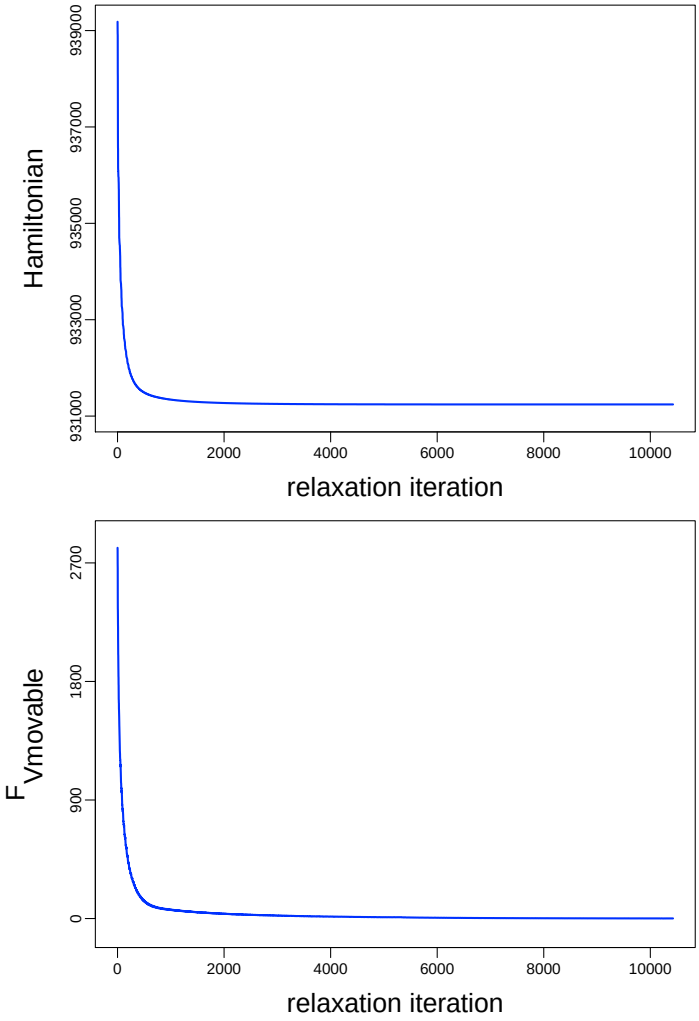
$$F_{\mathcal{V}_{\text{movable}}} = \sum_{v \in \mathcal{V}_{\text{movable}}} \|F_v\| =$$

$$= \sum_{v \in \mathcal{V}_{\text{movable}}} \| \sum_{e=u \sim v} k[e](\ell[e] - \ell_0[e]) \frac{\bar{x}_v - \bar{x}_u}{\ell[e]} \|, \tag{4.1}$$

where $\|\cdot\|$ is the function that assigns to a vector in the Euclidean space $\mathbb{R}^3$ its length.

A plot of values of $F_{\mathcal{V}_{\text{movable}}}$ during following iterations of the relaxation algorithm is presented in the bottom figure in 4.1.

As we can see in Figure 4.1, during the relaxation process, function $F_{\mathcal{V}_{\text{movable}}}$ asymptotically converges to 0. In the same time, the Hamiltonian converges to its local minimum.



a) The change of the value of the Hamiltonian function during following iterations of the relaxation algorithm.

b) The change of the value of $F_{\mathcal{V}_{\text{movable}}}$ during following iterations of the relaxation algorithm.

Figure 4.1: Dynamics of the relaxation procedure

4.4 Properties of parametric learning algorithm

In this section, we present main aspects of the dynamics of the parametric learning algorithm (see Section 2.3). In Subsection 4.4.1, we show how the error Φ changes in sequential epochs of the learning process when the system is adapted with and without noise. Next, in 4.4.2, we define a condition which has to be satisfied to stop the learning process.

4.4.1 Dynamics of error Φ during adaptation process

In this subsection, we show how the error function Φ changes while the learning algorithm is running. We expect that the trend of the function decreases.

During the learning phase, the profile of the Hamiltonian 2.12 for each given training example continuously changes. Let us denote the equilibrium state $G[\bar{x}^0_{\mathcal{V}_{\text{movable}}}, \bar{y}^{(i)}_{\mathcal{V}_{\text{con}}}]$, which was calculated for the i^{th} training example at the k^{th} epoch of the parametric learning algorithm 2, as $G[\bar{y}^{(i)}_{\mathcal{V}_{\text{con}}}]^k$. During the adaptation process, $G[\bar{y}^{(i)}_{\mathcal{V}_{\text{con}}}]^{k+1}$ might be a minimum which has not evolved from $G[\bar{y}^{(i)}_{\mathcal{V}_{\text{con}}}]^k$. It happens, when the movable nodes of the system were driven to a configuration, which is separated from the minimum evolved from $G[\bar{y}^{(i)}_{\mathcal{V}_{\text{con}}}]^k$ by a potential barrier impassable for the gradient descent dynamics. The system may in the forthcoming epochs return to the minimum that evolved from $G[\bar{y}^{(i)}_{\mathcal{V}_{\text{con}}}]^k$. If not, the parametric learning algorithm will evolve successive minima $(G[\bar{y}^{(i)}_{\mathcal{V}_{\text{con}}}]^{k+2}, G[\bar{y}^{(i)}_{\mathcal{V}_{\text{con}}}]^{k+3}, G[\bar{y}^{(i)}_{\mathcal{V}_{\text{con}}}]^{k+4}, \ldots)$ from $G[\bar{y}^{(i)}_{\mathcal{V}_{\text{con}}}]^{k+1}$ and adjust spring parameters of the system to minimise $\Phi^{(i)}$ for them. In both cases, the value of Φ temporarily increases, sometimes very significantly.

In Figures 4.2 and 4.3 we can see how the error Φ changes during following epochs of the learning process. During the learning process the value of the error function Φ, despite of a lot of oscillations, has a decreasing trend. Even after rarely occurring large peaks, the spring system quickly returns to already developed adaptation. A noise factor does not change noticeably frequency of peaks of the error function. Tests results confirm justness of application of the parametric learning algorithm to find the solution of a mechanical behaviour problem. A global minimum of Φ is unknown, so we can not tell the result is a global test solution.

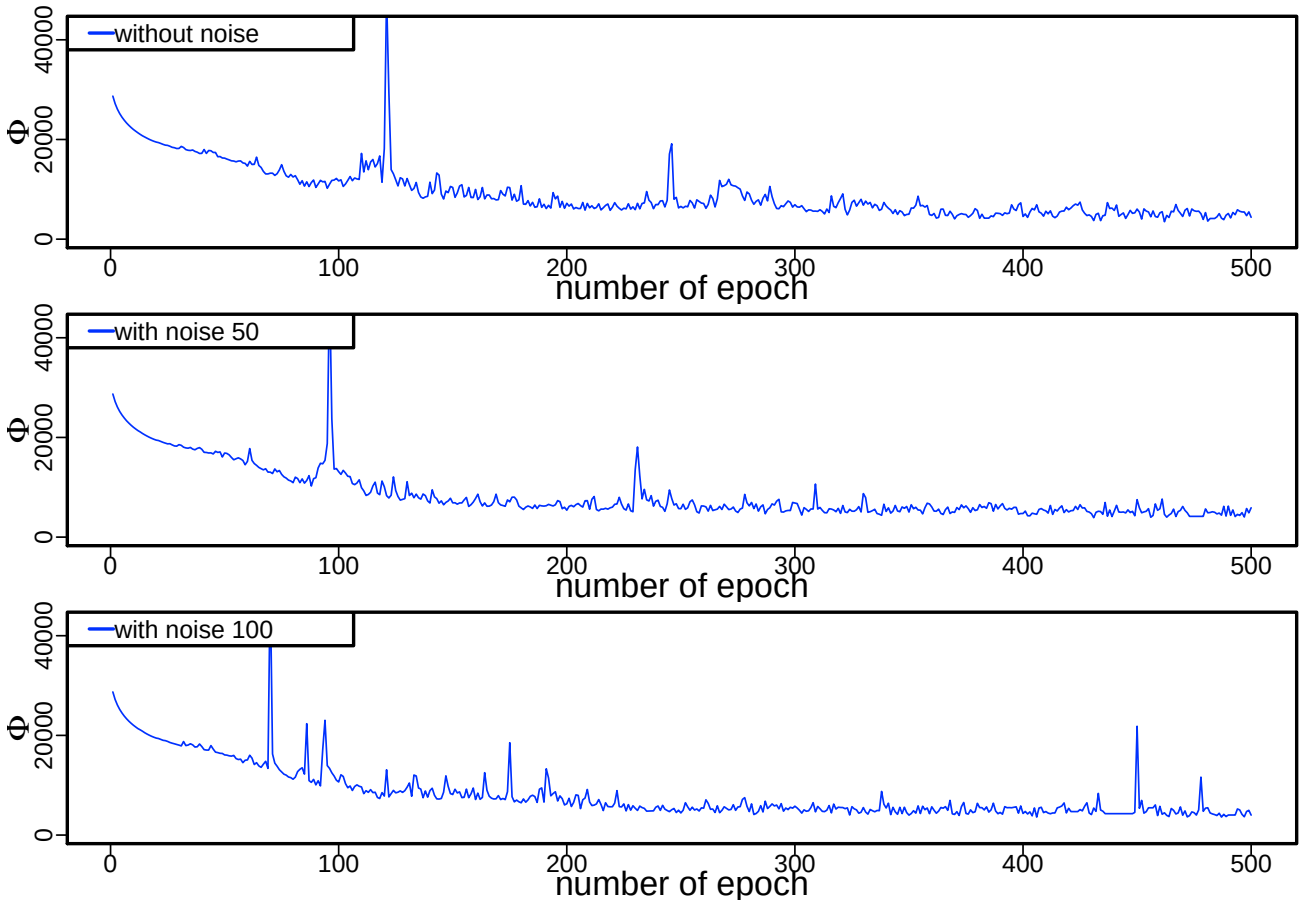


Figure 4.2: The value of the error function Φ vs the number of epochs of the parametric learning algorithm. For each system $|\boldsymbol{\mathcal{V}}_*| = \mathbf{10}$, $|\mathcal{V}_{\text{fixed}}| = 2$, $|\mathcal{V}_{\text{con}}| = 7$, $|\mathcal{V}_{\text{obs}}| = 10$, $c_{edge} = 8$, $N = 3$.

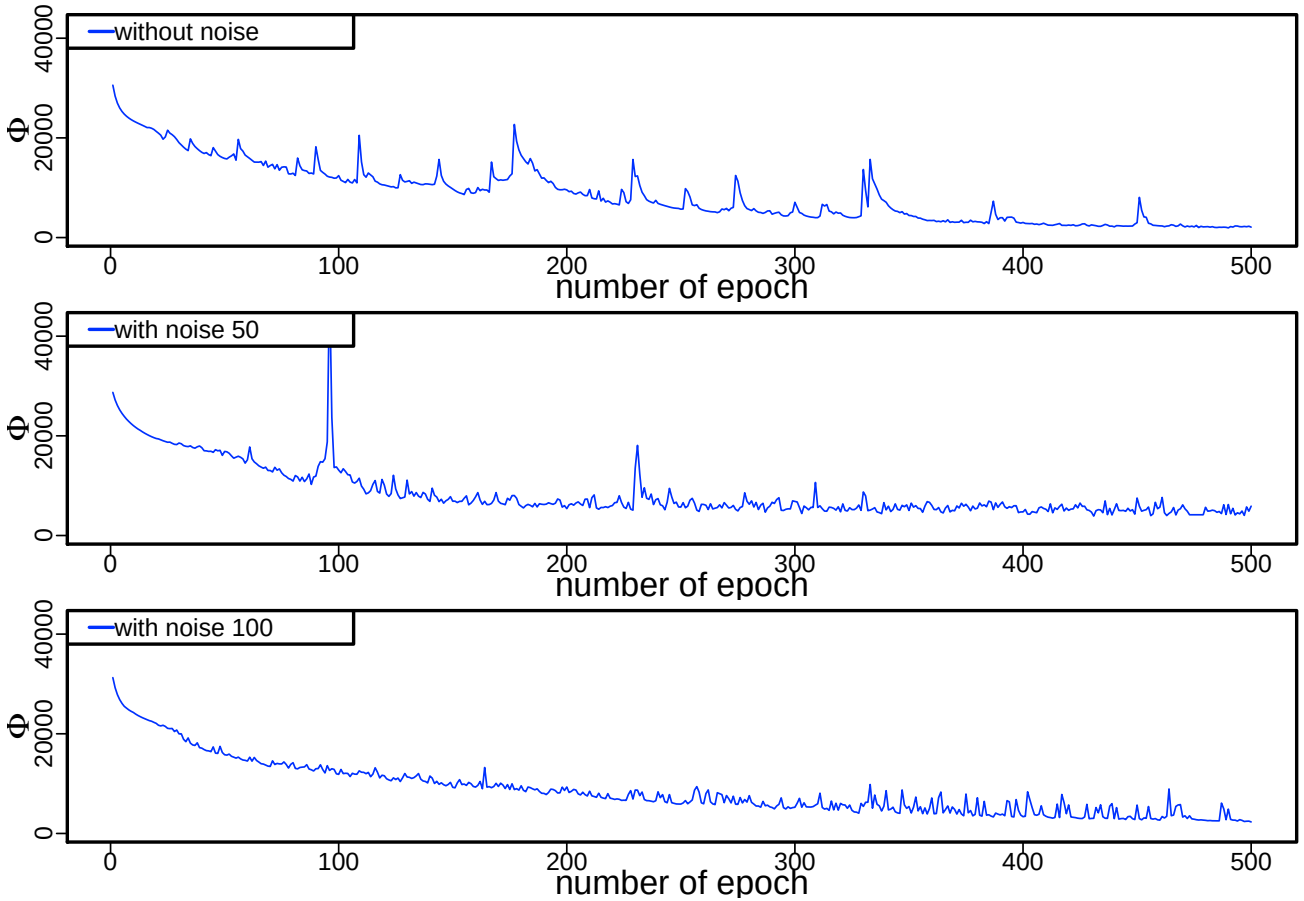


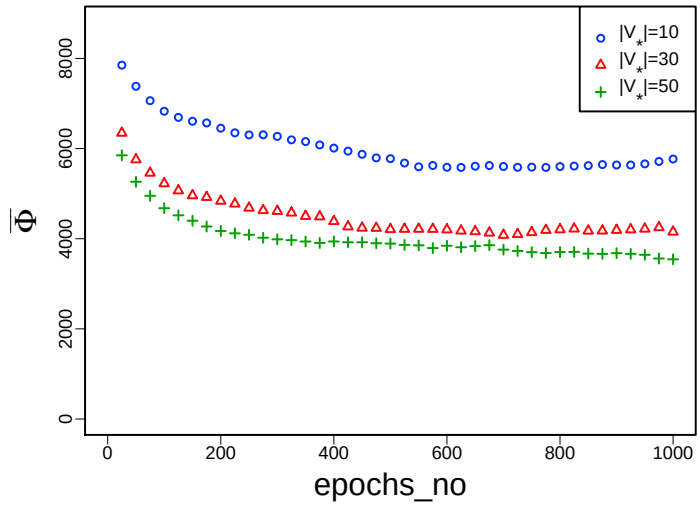
Figure 4.3: The value of the error function Φ vs the number of epochs of the parametric learning algorithm. For each system $|\boldsymbol{\mathcal{V}}_*| = \mathbf{50}$, $|\mathcal{V}_{\text{fixed}}| = 2$, $|\mathcal{V}_{\text{con}}| = 7$, $|\mathcal{V}_{\text{obs}}| = 10$, $c_{edge} = 8$, $N = 3$.

4.4.2 Stop condition

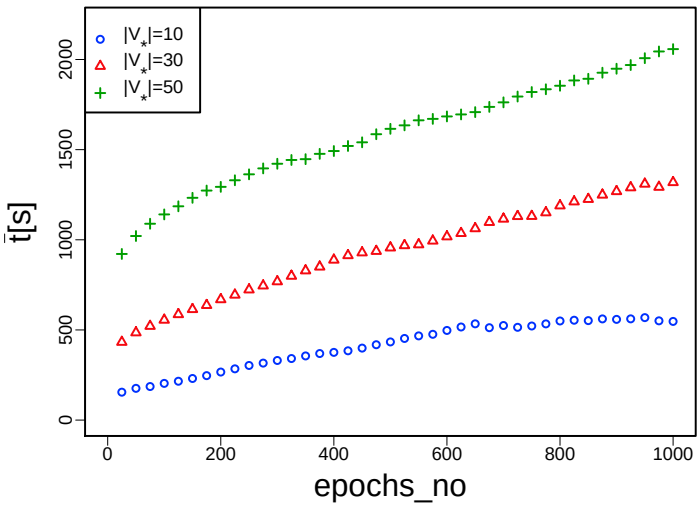
The parametric learning algorithm stops its execution, after a predefined number of epochs executed in a row without adaptation improvement (without decrease of the value of the error Φ). Such maximal number of permitted, sequential, and not boosting iterations is defined by a parameter *epochs_no*. In this subsection, we explore influence of *epochs_no* on time of learning execution and the quality of adaptation.

To conduct tests we constructed spring systems $\mathcal{G}$ along with the set of 3 training examples as described in Section 4.2. Results of tests are depicted in Figure 4.4.

As we see, the increase of the parameter *epochs_no* results in better adaptation of trained systems, at the expense of time consumption. Initially, the error Φ decreases abruptly, while *epochs_no* grows up, but with time it is suppressed. In all our working examples we stop adaptation after 150 epochs with non improving error Φ. It does not give the best adaptation, but this value is sufficient for testing numeric properties of the learning algorithm in acceptable time.



a) The plot over 100 tests of the average error $\bar{\Phi}$ versus applied *epochs_no* in simulations.



b) The plot over 100 tests of the average time versus applied *epochs_no* in simulations.

Figure 4.4: Dynamics for each size of auxiliary nodes set $|\mathcal{V}_*| \in \{10, 30, 50\}$ was calculated by averaging over 100 simulations. For each system $|\mathcal{V}_{\text{fixed}}| = 2,\ |\mathcal{V}_{\text{con}}| = 7,\ |\mathcal{V}_{\text{obs}}| = 10,\ c_{edge} = 4,\ N = 3$.

4.5 Number of resources of a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$

In this section, we test how the number of resources of a graph affects on numerical properties of the parametric learning algorithm. In Subsection 4.5.1, we specify synthetic problems which are solved by the parametric learning algorithm, and we illustrate numerical results of performed tests. We summarise outcomes of tests in 4.5.2.

4.5.1 Explored cases

We test 3 cases with the various level of complexity of the training examples set. Namely, they have a various number of control/observed nodes, training examples N, and various size of scattering of locations of nodes $v \in \mathcal{V}_{\text{con}} \cup \mathcal{V}_{\text{obs}}$ in their training examples (various values of parameters θ_1, θ_2 are applied during construction of sets $(E^{(i)})_{i=1}^{N}$, see 4.2.1).

We present plots with following statistics for each subsequent case:

1. The mean value of the **error Φ** against the number of **auxiliary vertices $|\mathcal{V}_*|$** for various parameters c_{edge}.

2. The mean **time consumed by the simulations** against the number of **auxiliary vertices $|\mathcal{V}_*|$** for various parameters c_{edge}.

3. The mean value of the **error Φ** against the number of **edges $|\mathcal{E}|$** for various parameters c_{edge}.

4. The mean **time consumed by the simulations** against the number of **edges $|\mathcal{E}|$** for various parameters c_{edge}, $|\mathcal{E}| \approx c_{edge} \cdot |\mathcal{V}|$.

In Table 4.1 we define three training problems solved during tests. These are simple training problem, with multiple training examples, and with multiple observed nodes, respectively.

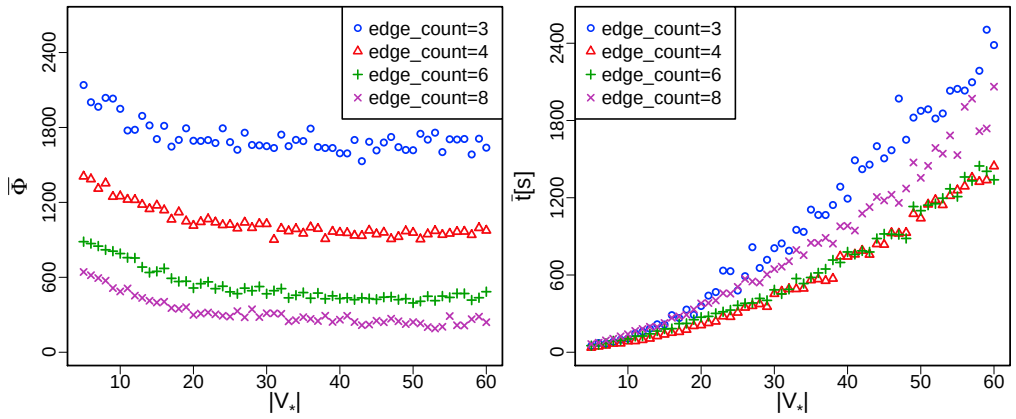
case number	$\lvert\mathcal{V}_{\mathbf{con}}\rvert$	$\lvert\mathcal{V}_{\mathbf{obs}}\rvert$	$\boldsymbol{N}$ number of training examples	$\boldsymbol{\theta_1}$ and $\boldsymbol{\theta_2}$	initial order of $\boldsymbol{\Phi}$
case 1	5	5	4	40	$8 \cdot 10^3$
case 2	5	5	10	40	$9 \cdot 10^3$
case 3	10	15	3	30	$3 \cdot 10^3$

Table 4.1: Parameters which define three training problems.

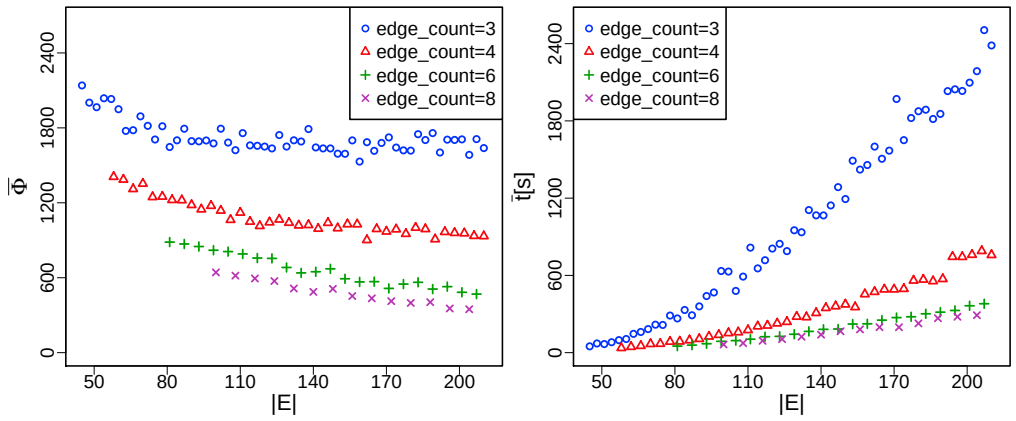
For each problem given in Table 4.1 we generate and train spring systems 100 times for the following resources:

- $c_{edge} \in \{3, 4, 6, 8\}$,
- $|\mathcal{V}_*| \in \{5, \ldots, 60\}$ for cases 1 and 2, $|\mathcal{V}_*| \in \{5, \ldots, 50\}$ for case 3,
- $|\mathcal{V}_{\text{fixed}}| = 2$.

During the learning phase, a noise is introduced to the system (see Section 2.5). For cases 1 and 2 we use $noise\_radius = 100$ and for case 3 we use $noise\_radius = 50$. Results for cases $1, 2, 3$ are presented in Figures 4.5, 4.6, 4.7, respectively.

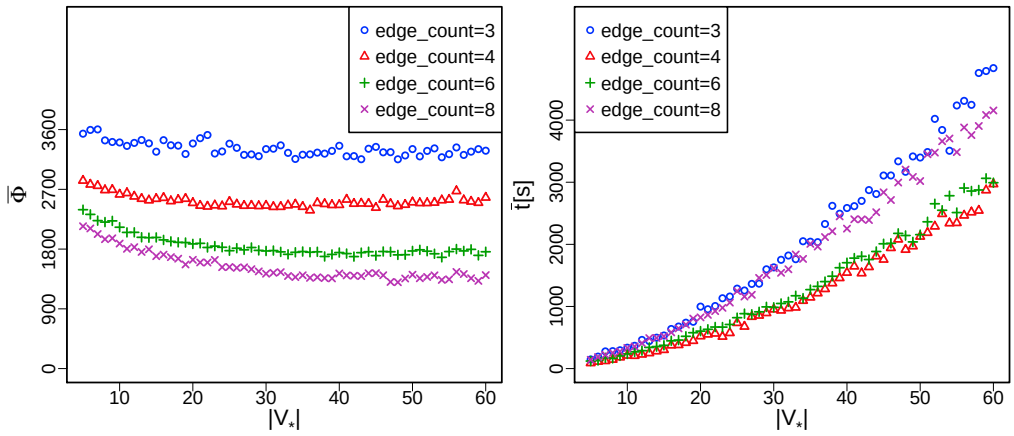


(a) The plot of $\bar{\mathbf{\Phi}}$ (left) and the mean **time of simulations** (right) versus the number of **auxiliary vertices** $|\boldsymbol{\mathcal{V}}_*|$

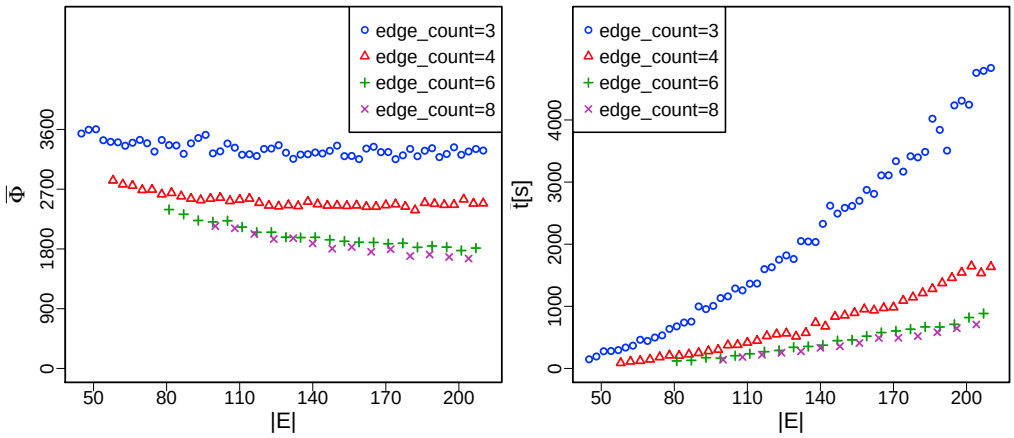


(b) The plot of $\bar{\mathbf{\Phi}}$ (left) and the mean **time of simulations** (right) versus the number of **edges** $|\boldsymbol{E}|$

Figure 4.5: Each point was calculated by averaging over 100 simulations. For each system $|\boldsymbol{\mathcal{V}}_{\mathbf{con}}| = \mathbf{5}$, $|\boldsymbol{\mathcal{V}}_{\mathbf{obs}}| = \mathbf{5}$, $\boldsymbol{N} = \mathbf{4}$, $|\mathcal{V}_{\text{fixed}}| = 2$, $\theta_1 = 40$, $\theta_2 = 40$.

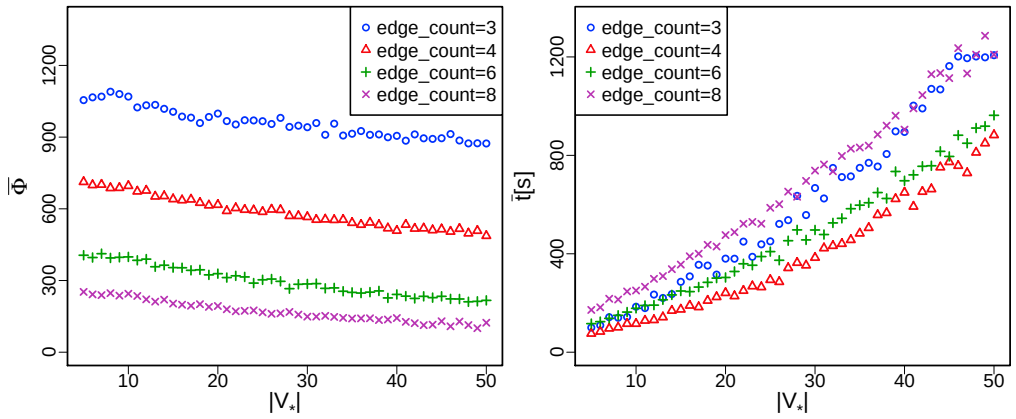


(a) The plot of $\bar{\mathbf{\Phi}}$ (left) and the mean **time of simulations** (right) versus the number of **auxiliary vertices** $|\boldsymbol{\mathcal{V}}_*|$

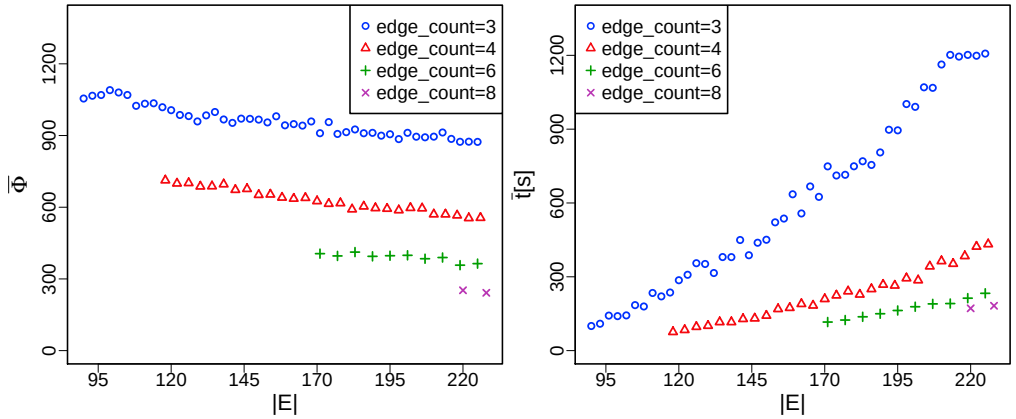


(b) The plot of $\bar{\mathbf{\Phi}}$ (left) and the mean **time of simulations** (right) versus the number of **edges** $|\boldsymbol{E}|$

Figure 4.6: Each point was calculated by averaging over 100 simulations. For each system $|\boldsymbol{\mathcal{V}}_{\mathbf{con}}| = \mathbf{5}$, $|\boldsymbol{\mathcal{V}}_{\mathbf{obs}}| = \mathbf{5}$, $\boldsymbol{N} = \mathbf{10}$, $|\mathcal{V}_{\text{fixed}}| = 2$, $\theta_1 = 40$, $\theta_2 = 40$.



(a) The plot of $\bar{\mathbf{\Phi}}$ (left) and the mean **time of simulations** (right) versus the number of **auxiliary vertices** $|\boldsymbol{\mathcal{V}}_*|$



(b) The plot of $\bar{\mathbf{\Phi}}$ (left) and the mean **time of simulations** (right) versus the number of **edges** $|\boldsymbol{E}|$

Figure 4.7: Each point was calculated by averaging over 100 simulations. For each system $|\boldsymbol{\mathcal{V}}_{\mathbf{con}}| = \mathbf{10}$, $|\boldsymbol{\mathcal{V}}_{\mathbf{obs}}| = \mathbf{15}$, $\boldsymbol{N} = \mathbf{3}$, $|\mathcal{V}_{\text{fixed}}| = 2$, $\theta_1 = 30$, $\theta_2 = 30$.

In Figure 4.8 boxplots[1] depict results for respective training cases (sequentially for easy case to solve, with many training examples and with many observed nodes). Each box represents distribution of $\bar{\Phi}$ for a given c_{edge}.

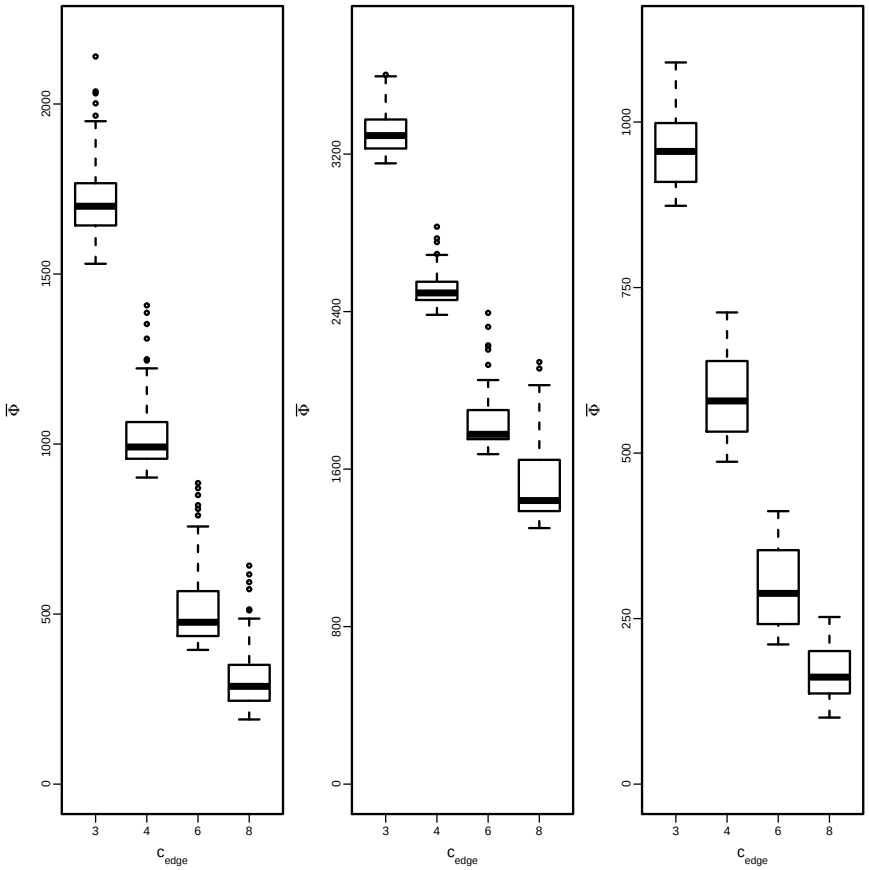


Figure 4.8: Boxplots for respective training cases. Each box represents $\bar{\Phi}$ for a given c_{edge}.

[1]Boxplot is a method for graphically depicting groups of numerical data through their quartiles. Box vertical edges plots the first Q_1 and the third quartiles Q_3, and the band inside the box is the median. Lines going out vertically from the boxes (whiskers) extend to the most extreme data point which is no more than 1.5 times the interquartile range $(Q_3 - Q_1)$ from the box.

4.5.2 Conclusions

Our tests imply that the number of graph resources has a significant effect on the learning ability of spring systems. It can be seen in the plots depicting $\bar{\Phi}$ against $|\mathcal{V}_*|$ in the left figures of 4.5a, 4.6a, and 4.7a. Both, extension of the size of the auxiliary nodes set $\mathcal{V}_*$ and increment of the value of the parameter c_{edge}, asymptotically decrease the value of $\bar{\Phi}$. The lower bound obtained by the mean value of Φ depends on complexity of a learning problem. The more complex the problem is, the higher error.

In order to explore influence of the size of the set $\mathcal{V}_*$ on the learning ability of the system we present the mean error $\bar{\Phi}$ for $|\mathcal{V}_*| = 5$ and $|\mathcal{V}_*| = 50$. Next, for these two statistics we compute their $ratio = \frac{\bar{\Phi}\ for\ |\mathcal{V}_*|=5}{\bar{\Phi}\ for\ |\mathcal{V}_*|=50}$. These results are presented in Table 4.2. For $c_{edge} = 8$ the ratio is equal to **2.66**, **1.52** and **2.04** for corresponding cases of training categories.

case	c_{edge}	$\|\mathcal{V}_*\| = 5$	$\|\mathcal{V}_*\| = 50$	$ratio$
1	3	2139	**1618**	1.32
1	4	1407	958	1.47
1	6	885	395	2.24
1	8	642	**241**	2.66
2	3	3539	**3307**	1.07
2	4	2831	2504	1.13
2	6	2393	1772	1.35
2	8	2144	**1414**	1.52
3	3	1054	**873**	1.20
3	4	712	487	1.46
3	6	405	216	1.88
3	8	252	**123**	2.04

Table 4.2: The following columns present $\bar{\Phi}$ for spring systems with various number of auxiliary nodes for cases 1 and 2, and 3. Next, $ratio = \frac{\bar{\Phi}\ for\ |\mathcal{V}_*|=5}{\bar{\Phi}\ for\ |\mathcal{V}_*|=50}$ is calculated.

Table 4.3 presents influence of the value of the parameter c_{edge} on the learning ability of spring systems. It can be strictly measured by the ratio of the mean error Φ for spring systems built with parameters $c_{edge} \in \{3, 8\}$ and with the set of auxiliary nodes $|\mathcal{V}_*| = 50$. The increment of the parameter c_{edge} from 3 to 8 improves learning ability **6.71**, **2.34**, and **7.10** times for the respective training problems. So, we can conclude that the increment of the parameter c_{edge} has a larger effect on enhancement of learning capability of spring systems than augmentation of the set of auxiliary nodes $\mathcal{V}_*$.

case	$\boldsymbol{\|\mathcal{V}_*\| = 50}$, $\boldsymbol{c_{edge} = 3}$	$\boldsymbol{\|\mathcal{V}_*\| = 50}$, $\boldsymbol{c_{edge} = 8}$	$\boldsymbol{ratio}$
1	1618	241	6.71
2	3307	1414	2.34
3	873	123	7.10

Table 4.3: The following columns present the mean errors, which are calculated for $|\mathcal{V}_*| = 50$ for cases 1, 2, and 3. Next, $ratio = \frac{\bar{\Phi}\ for\ |\mathcal{V}_*|=50,\, c_{edge}=3}{\bar{\Phi}\ for\ |\mathcal{V}_*|=50,\, c_{edge}=8}$ is presented.

Average time necessary to yield the error by respective networks, is presented in the right plots of Figures 4.5, 4.6, and 4.7. Clearly, the more auxiliary nodes a network has, the longer it takes to train it. As we can see in the right plots in Figures 4.5a, 4.6a, and 4.7a the mean time of learning execution grows faster than linearly with the size of $\mathcal{V}_*$.

In the plots of Figures 4.5b, 4.6b, and 4.7b we can observe that for systems with similar number of edges, these with the higher parameter c_{edge} and consequently with the lower number of auxiliary nodes, yield significantly lower values of the mean error of Φ (left figures) and lower time consumption (right figures). That implies that the size of the set of edges is not so important as an average degree of nodes in a graph.

Table 4.4 presents how changes the order of the error Φ before and after

the learning processes for spring systems with $c_{edge} = 8$ and auxiliary nodes $|\mathcal{V}_*| = 50$.

case	initial order of $\mathbf{\Phi}$	final order of $\mathbf{\Phi}$	**ratio**
1	$8 \cdot 10^3$	$2 \cdot 10^2$	40
2	$9 \cdot 10^3$	$1 \cdot 10^3$	9
3	$3 \cdot 10^3$	$1 \cdot 10^2$	30

Table 4.4: The error Φ change during training processes. The last column presents the ratio for $\bar{\Phi}$ error before and after learning processes.

4.6 Energy profile and noise factor properties

In this section, we evaluate how adapted spring systems are resilient to convergence to local minima other than the trained ones. Each explored system was trained with and without noise.

All systems evaluated in this section have the following complexity of the training example set: $|\mathcal{V}_{\text{con}}| = 7$, $|\mathcal{V}_{\text{obs}}| = 10$, and the number of the training examples N is equal to 3.

For this problem we generate and train spring systems 100 times for each combination of the following resources the existence/magnitude of a noise factor: $c_{edge} \in \{4, 8\}$, $|\mathcal{V}_*| \in \{5, \ldots, 60\}$, $|\mathcal{V}_{\text{fixed}}| = 2$, and for discarded noise, with $noise\_radius = 50$ or with $noise\_radius = 100$.

For each adapted spring system, we sample two training examples, and we calculate the error Ψ after transition between them as described in 2.5.2 according to two scenarios:

- we sample 200 times training examples, and we perform transitions between their control nodes positions along semicircles. It gives us

20000 values of Ψ for 100 simulations for each combination of the spring system resources and the existence/magnitude of a noise factor.

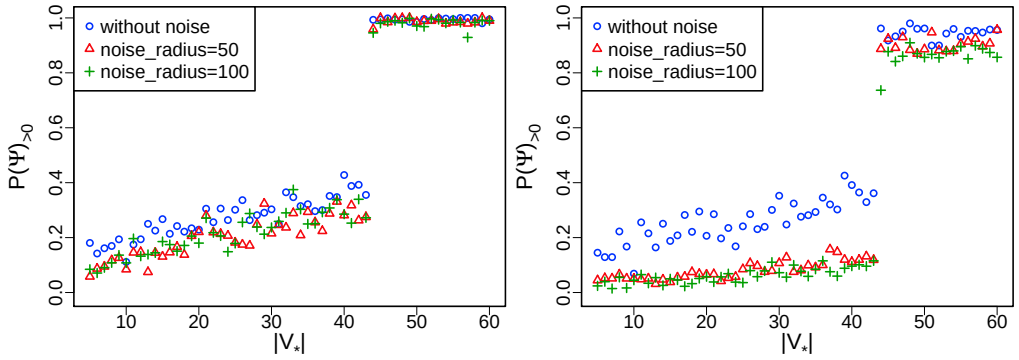
- for each pair of training examples we perform transitions between their control nodes positions along straight line $|N| \cdot |N-1| = 6$ times. It gives us 600 values of Ψ for 100 simulations for each combination of the spring system resources and the existence/magnitude of a noise factor.

If observed nodes fail to reach corresponding target positions, then $\Psi > 0$. For each combination of the spring system resources, the existence/magnitude of a noise factor, and separately for semicircle and straight line transitions, we calculate $P(\Psi_{>0})$. This is a ratio of Ψ values not equal to 0. So, this is chance that at least one of the observed nodes goes into a wrong position.

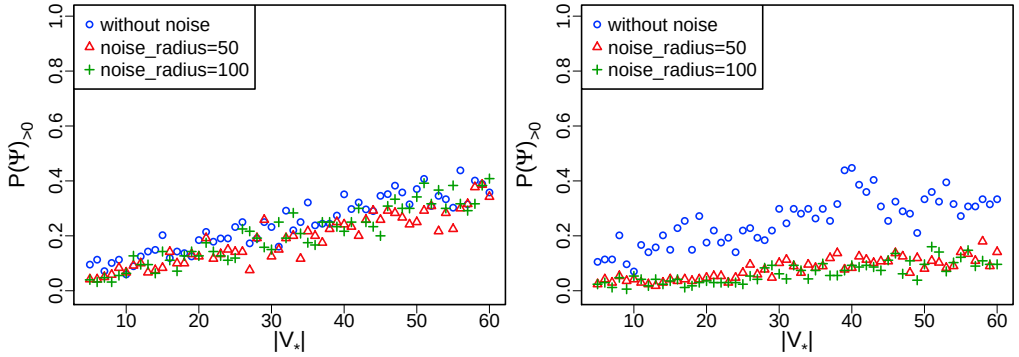
In Figure 4.9 we present $P(\Psi_{>0})$ against the number of auxiliary nodes $|\mathcal{V}_*|$. The plots show that spring systems trained with a noise and with a larger average degree of nodes are more resilient to the node perturbations. It is shown by the smaller value of statistics $P(\Psi_{>0})$ for such cases. The value of $P(\Psi_{>0})$ goes up together with $|\mathcal{V}|$. For transitions of control nodes along semicircle curves, increment of the size from $|\mathcal{V}| = 62$ to $|\mathcal{V}| = 63$ ($|\mathcal{V}_*| = 43$ to $|\mathcal{V}_*| = 44$) is critical for the Hamiltonian function profile. Spring systems which have the size of $\mathcal{V}$ less or equal to 62, have a small value of $P(\Psi_{>0})$, on the contrary for the size over this point, $P(\Psi_{>0})$ jumps to 1.

Also for spring systems adapted for the tests in previous Section 4.5, such transitions always happen when $|\mathcal{V}|$ is extend from 62 to 63. Alternation of factors like for example ratio of $|\mathcal{V}_{\text{frozen}}|$ to $|\mathcal{V}_{\text{movable}}|$ or number of training examples does not change it.

For transition of control nodes along line curves no critical point is observed. We conclude that big concentration of nodes with the low average graph distance generates spring systems with explosion of many undesirable local minima of the Hamiltonian function.



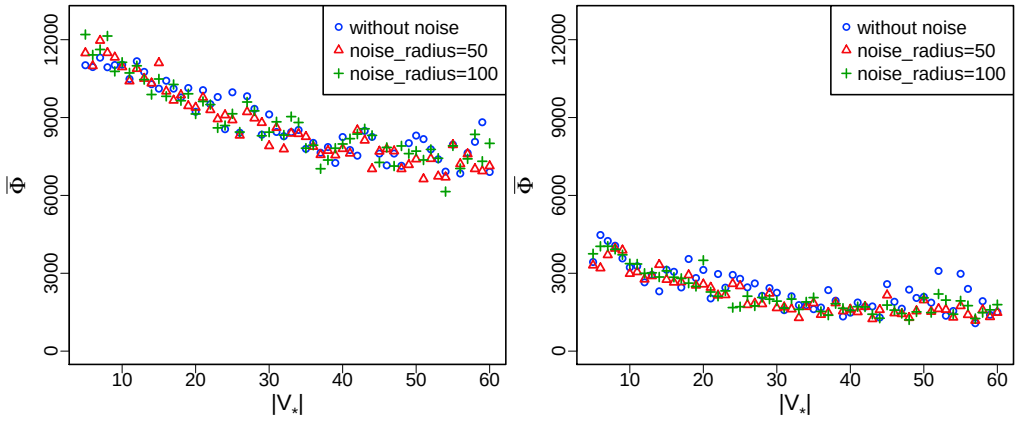
(a) Transitions of control nodes performed along **semicircle curves**



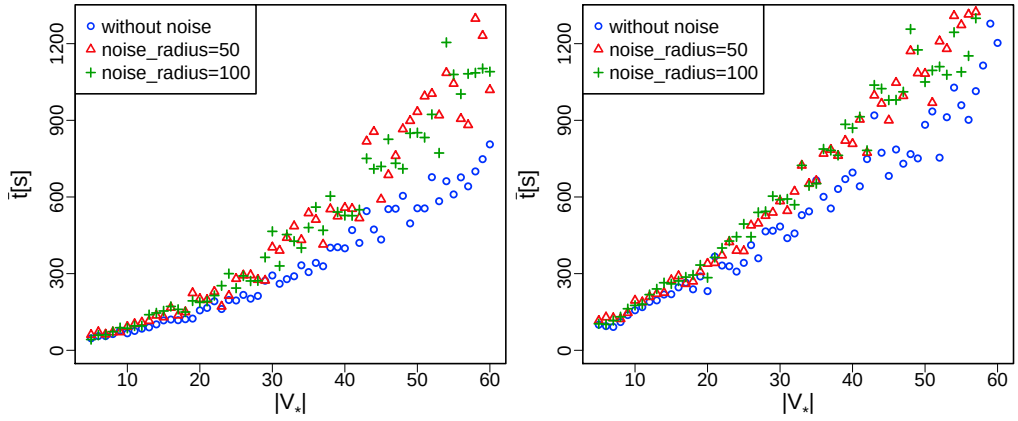
(b) Transitions of control nodes performed along **straight lines**

Figure 4.9: Plots present $\boldsymbol{P(\Psi_{>0})}$ against the number of auxiliary vertices $\boldsymbol{|\mathcal{V}_*|}$. In the left for $\boldsymbol{c_{edge} = 4}$ and in the right for $\boldsymbol{c_{edge} = 8}$. For each system $|\mathcal{V}_{\text{fixed}}| = 2,\ |\mathcal{V}_{\text{con}}| = 7,\ |\mathcal{V}_{\text{obs}}| = 10,\ N = 3$.

In Figure 4.10 we present the mean error and the mean time of simulation against $|\mathcal{V}_*|$.



(a) The plot of $\bar{\mathbf{\Phi}}$ against the number of auxiliary vertices $|\boldsymbol{\mathcal{V}}_*|$



(b) The plot of the **mean time of simulations** against $|\boldsymbol{\mathcal{V}}_*|$

Figure 4.10: In the left plots for $\boldsymbol{c_{edge} = 4}$ and in the right plots for $\boldsymbol{c_{edge} = 8}$. For each system $|\mathcal{V}_{\text{fixed}}| = 2,\ |\mathcal{V}_{\text{con}}| = 7,\ |\mathcal{V}_{\text{obs}}| = 10,\ N = 3$.

4.7 Conclusions

In this chapter, we have presented the most important numerical properties of the parametric learning algorithm.

In Section 4.3, we have shown how the value of the Hamiltonian function decreases during the following iterations of the relaxation procedure. In turn, in Section 4.4, we have depicted how the parametric learning algorithm iteratively finds better solutions (its quality is measured by the value of the error Φ) for a given problem.

In Section 4.5, we have shown that the mean error, to which the learning process converges, depends mainly on the topology of a trained spring system. A spring system obtains the best adaptation for a structure with a high average degree of nodes. The size of the set $\mathcal{V}_*$ should be kept small since increasing the number of its auxiliary nodes quickly stops improving the learning ability of the spring system. Moreover, the mean time consumed by the learning process grows up faster than linearly with the size of $\mathcal{V}_*$. Systems with the described topology are more resilient to the disturbance of locations of control nodes by external forces as has been shown in 4.6. This property is also improved by adding a noise factor.

Chapter 5

Conclusions

5.1 Summary

In this dissertation, we have proposed a new parametric learning algorithm for mechanical behaviour tasks carried out by complex spring systems (see Subsection 2.3.2). For a given collection of examples specifying target relationship between the control and observed nodes, the procedure outputs physical parameters (lengths, elastic constants) of a system designed to represent this relationship. We have implemented our algorithm and tested it using various examples. We have shown that it gives reasonable results for spring structures with a high average degree of nodes and with not to many auxiliary nodes (see 2.2.1 for definition of various types of nodes). Keeping a high average degree of nodes and a restricted number of auxiliary nodes, has additional advantages. It makes the system more resilient to a noise, which can cause wrong mapping of control onto observed nodes. Explored spring systems were constructed by our algorithm generating graph topology (see Section 2.4).

We have applied our model to mimic dynamics of proteins, see Chapter 3. The quality of the obtained results is very high. We have plotted the mean

value of forces as a function of a spring length for systems in their stable equilibria representing real-world configurations of proteins. We have shown that the obtained curve fits well to the shape of the Lennard-Jones force.

5.2 Further research

The algorithm, which trains spring systems, presented in this work, can be developed in various directions according to particular scientific problems.

It would be interesting to build spring systems representing more elaborated shapes like for example parts of a car body or an atypical building architecture. Found systems could form a basis to construct real objects with appropriate elastic properties. In other words, instead of checking if a given manually created spring system is appropriate to use, our model is employed to construct a spring system from the scratch. Hence, our method could support a computer-aided design with a useful utility. The constructed spring systems should also have a possibly simple structure in order to easily produce their real-world implementation.

After conducting the proposed research, our model could be a plugin of a computer-aided design (CAD) software in the areas like mechanical or structural engineering. Such tools get as an input a physical object with known elastic properties and explore whether or not it reacts in a target manner under the influence of physical external forces. If it distorts in a wrong way, it could be a good idea to apply our algorithm to adjust physical properties of the engineering structure. For such a purpose the learning algorithm proposed by us can only modify elastic constants (without lengths) if advisable.

Appendix A

Rigid graphs

In this chapter, we present the theory concerning rigid graphs (see Sections A.1 and A.2). In Section A.3, we present an algorithm building a generically minimally rigid graph. In Section A.4, we encapsulate how we remove degrees of freedom from structures of spring systems and why it is necessary.

A.1 Rigid graph

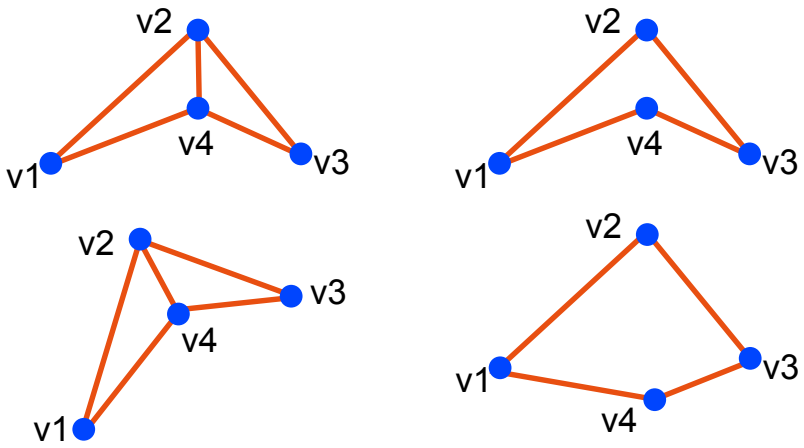
One of the first results regarding rigidity theorem were provided by Cauchy in 1813, see [9]. Although rigidity problems were of immense interest of engineers, a strong mathematical study of these kinds of problems has occurred relatively recently. Here we present a theory concerning this area of knowledge, which is relevant in this dissertation.

Let us assume that we are given an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with a finite vertex/node set $\mathcal{V}$ and an edge/spring set $\mathcal{E}$ (see Section 2.2). In the context of theory presented in this appendix section, vertices $\mathcal{V}$ are embedded in $\mathbb{R}^d$ and $d = 3$.

Definition A.1.1. ***(after [3, 6, 29])*** *The graph $\mathcal{G}$ along with its embedding in the three-dimensional Cartesian space $\bar{x}_{\mathcal{V}} = ((\bar{x}_v)_{v\in\mathcal{V}}) \in \mathbb{R}^{d\cdot|\mathcal{V}|}$ called*

configuration, is said to be rigid in $\mathbb{R}^d$ *if and only if every continuous motion of vertices* $\mathcal{V}$*, beginning at* $\bar{x}_{\mathcal{V}}$ *and preserving the lengths for all edges* $\ell[e]$*,* $e \in \mathcal{E}$*, terminates at a configuration* $\bar{x}'_{\mathcal{V}} = ((\bar{x}'_v)_{v\in\mathcal{V}})$ *which is the image* $T\bar{x}_{\mathcal{V}} = ((T\bar{x}_v)_{v\in\mathcal{V}})$ *under an isometry* T *of* $\mathbb{R}^d$*. Moreover, we say that the configuration* $\bar{x}_{\mathcal{V}}$ *of the graph is rigid. Otherwise, the graph* $\mathcal{G}$ *(configuration) is flexible in* $\mathbb{R}^d$*.*

Figure A.1 presents examples of a rigid (left) and flexible (right) graph in $\mathbb{R}^2$ and in Figure A.2 is an example of a rigid graph in $\mathbb{R}^3$.



(a) A rigid graph in $\mathbb{R}^2$ (b) A flexible graph in $\mathbb{R}^2$

Figure A.1: Examples of rigid (left) and flexible (right) graph in $\mathbb{R}^2$. For both cases two configurations are presented. The first one is derived from the second one by continuous motions of vertices, preserving the lengths for all edges. For the rigid graph, the distances between all pairs of nodes were preserved, for the flexible graph were not.

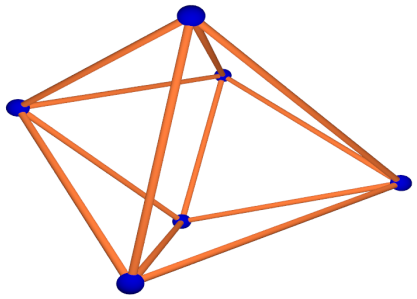


Figure A.2: A rigid graph in $\mathbb{R}^3$.

A.2 Generic rigidity

Recently, it has been proved that determining whether or not a graph is rigid for $d > 2$ is NP-hard, see [1]. This problem becomes more tractable if we assume that the graph $\mathcal{G}$ along with configuration is *generic* i.e. there are no algebraic dependencies between the coordinates $(\bar{x}_v)_{v\in\mathcal{V}}$, see [8, 19]. Then it is known that the rigidity of the graph depends only on its topology. The generic configurations of a given graph $\mathcal{G}$ are either all rigid or all flexible in $\mathbb{R}^d$. A graph $\mathcal{G}$ is called *generically rigid* in $\mathbb{R}^d$ if and only if all its generic configurations are rigid in $\mathbb{R}^d$. Otherwise, all its configurations are flexible and the graph $\mathcal{G}$ is called *generically flexible*, see [38]. A randomly constructed configuration in $\mathbb{R}^d$ with high probability is generic (the Lebesque's measure of the set of configurations which are not generic is equal to 0). Should it happen that the configuration is not generic (for instance it was arbitrary predefined), this property can be easily restored by small perturbation of positions of nodes. So, without loss of generality we can assume that the graph $\mathcal{G}$ has generic configuration and determine whether or not it is rigid based solely on its topology.

Definition A.2.1. *([2, 29, 38]) A (generically) rigid graph which after removing any of its edges becomes a (generically) flexible is called (generically) minimally rigid graph.*

It is easy to notice that the addition of extra edges to a generically rigid graph will not affect its generic behaviour. In consequence, it seems justified that in our numerical tests we construct spring systems by building a minimally rigid graph and by adding to it extra edges. Sometimes this approach might not be convenient and then we enforce sufficient rigidity of the structure by creating a graph with adequately a high average degree of nodes.

In 1970, Laman published a theorem that can be used to test whether or not a planar graph is rigid in $\mathbb{R}^2$.

Theorem (Laman, [24]) A planar graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is rigid for dimension 2 if and only if there is a subset $\mathcal{E}' \subseteq \mathcal{E}$ such that:

1. $|\mathcal{E}'| = 2|\mathcal{V}| - 3$,
2. for all $\mathcal{E}'' \subset \mathcal{E}'$ where $|\mathcal{V}(\mathcal{E}'')| \geq 2$ we have $|\mathcal{E}''| \leq 2|\mathcal{V}(\mathcal{E}'')| - 3$, where $\mathcal{V}(\mathcal{E}'')$ denotes all vertices $v \in \mathcal{V}$ for which exists at least one incident edge in $\mathcal{E}''$.

This condition is necessary and sufficient. The graph $\mathcal{G}' = (\mathcal{V}, \mathcal{E}')$ is a minimally rigid graph. By modifying Laman's condition we get the following statement for the three-dimensional space.

Theorem ([17]) A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is rigid for dimension 3 if and only if there is a subset $\mathcal{E}'$ of $\mathcal{E}$ such that:

1. $|\mathcal{E}'| = 3|\mathcal{V}| - 6$,
2. for all $\mathcal{E}'' \subset \mathcal{E}'$ where $|\mathcal{V}(\mathcal{E}'')| \geq 3$ we have $|\mathcal{E}''| \leq 3|\mathcal{V}(\mathcal{E}'')| - 6$.

Although this condition is necessary, it is no longer sufficient. In Figure A.3 is presented a graph flexible in 3d, which satisfies Laman's condition.

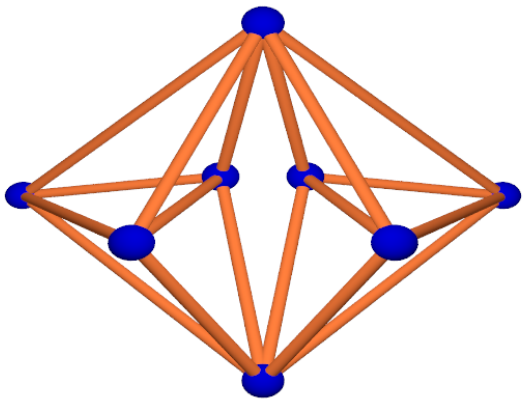


Figure A.3: A graph satisfies Laman's condition, but it is flexible.

A.3 Henneberg constructions

Henneberg construction is an inductive method, which is applied to create a generically minimally rigid graph. Here we briefly describe this method and its substantial properties. Theory presented in this section is thoroughly covered in Section 5 in [38].

Definition A.3.1. *Let us assume that we are given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ embedded in $\mathbb{R}^d$.*

- *The H_1^d (**vertex addition**) operation applied to $\mathcal{G}$, inserts one new vertex that gets connected to d existing ones.*
- *The H_2^d (**edge split**) operation applied to $\mathcal{G}$, replaces an edge by a new vertex that gets connected to its endpoints and additionally to $d-1$ other vertices.*

Definition A.3.2. *A Henneberg d-sequence for a graph $\mathcal{G}$ is a sequence of graphs $\mathcal{G}_1, \ldots, \mathcal{G}_n$ with the following properties:*

- *$\mathcal{G}_1 = K_{d+1}$, where K_{d+1} is so called a complete graph consisting of $d+1$*

vertices and in which every pair of distinct nodes is connected by a unique edge,

- $\mathcal{G}_n = \mathcal{G}$,
- $\forall_{i\in\{2,\ldots,n-1\}}$ $\mathcal{G}_{i+1}$ *is obtained from* $\mathcal{G}_i$ *through the* H_1^d *or* H_2^d *step.*

Theorem A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is generically minimally rigid in $\mathbb{R}^2$ if and only if can be generated by a Henneberg 2-sequence.

Theorem Every graph $\mathcal{G}$ obtained by Henneberg d-construction is generically minimally rigid in $\mathbb{R}^d$.

For $d > 2$ Henneberg construction does not enable us to construct all generically minimally rigid graphs in $\mathbb{R}^d$. However, in our dissertation we are mainly interested in building generically rigid graphs in $\mathbb{R}^3$ by constructing a generically minimally rigid graph and by adding extra edges. Henneberg construction is sufficient for us, moreover, we restrict ourselves to applying only H_1^3 operation. In Figure A.4 we present an example of a Henneberg 2-sequence, where the following graphs are obtained through H_1^2.

A.4 Degrees of freedom of rigid graph

A graph $\mathcal{G}$ in a three-dimensional space, for which $|\mathcal{V}|$ is equal to n and with an empty set of edges $\mathcal{E}$, has $3n$ degrees of freedom (all points can freely move in the space). By augmentation of the set $\mathcal{E}$ by adding sequential edges constituting connections of minimally rigid graph of $\mathcal{G}$, the number of degrees of freedom decreases. Eventually, every (minimally) rigid graph in a three-dimensional space has 6 degrees of freedom. The half of them arise from three independent coordinates, along which the graph can be translated, and remaining ones from three axes of rotation. For more details see Section 4 in

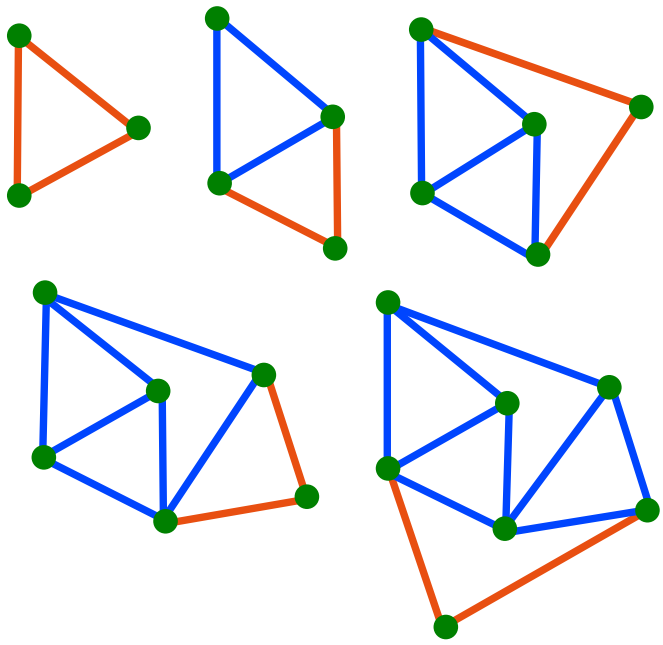


Figure A.4: An example of a Henneberg 2-sequence, where the following graphs are obtained through H_1^2.

[18]. It is vital to prevent a spring system from these rotations, translations or combinations of them, in order to avoid its erratic movements during the relaxation process. This is obtained by keeping 3 noncollinear nodes frozen, and in our model it is realised by the sets $\mathcal{V}_{\text{fixed}}$ and $\mathcal{V}_{\text{con}}$.

Appendix B

Pseudocode

In this appendix chapter we briefly list technical details of the learning procedure 2.3 and algorithms designed for spring systems. In the first Section B.1, we describe technical details and the pseudocode of the relaxation procedure. In Section B.2, we present the pseudocode of the parametric learning algorithm, for which various alternative stop conditions are described in Section B.3. Next, in B.4 we analyse influence of numerous parameters of the algorithm on the adaptation process. In Sections B.5 and B.6, we define pseudocodes of procedures which create a random structure of a spring system and a set of training examples if not given. In B.7 we provide technical description of the scheme adding a noise to positions of movable nodes during the learning process. In the end, in B.8 we provide the pseudocode of the algorithm, which evaluates robustness of adapted spring systems on perturbation of movable nodes positions.

B.1 Relaxation

In this section, we present the pseudocode of the procedure seeking an equilibrium of $\mathcal{G}$. The scheme moves nodes as given in the Algorithm 1.

Algorithm 1 Relaxation

1: **procedure** RELAXATION($\mathcal{G}$)
2: push all unbalanced, movable nodes to a First-In-First-Out queue Q
3: **while** Q is not empty **do**
4: pop an element q from the queue Q
5: $counter \leftarrow 0$
6: $whenStabilised \leftarrow -1$
7: **repeat**
8: $counter \leftarrow counter + 1$
9: calculate the vector of the force $\vec{f}$ acting on q
10: move q according to the formula $\bar{x}_q \leftarrow \bar{x}_q + step\_size \cdot \vec{f}$
▷ $step\_size$ is a small positive constant
11: **if** $\vec{f} = (f_1, f_2, f_3)$ satisfies $\forall_{i=1}^{3} |f_i| < threshold$ **then**
▷ we assume that the node q is in an equilibrium state
▷ $threshold$ is a small positive constant
12: $whenStabilised \leftarrow counter$
13: **end if**
14: **until** $max\_counter > counter$ and $whenStabilised = -1$
▷ $max\_counter$ is an integer constant greater than 0
15: **if** $whenStabilised = -1$ **then**
▷ the vertex q has not been stabilised sufficiently
16: push again q to the queue Q
17: **end if**
18: **if** $whenStabilised \neq 1$ **then**
▷ q was significantly relocated in the repeat loop 7
19: push all movable neighbours of q, which have not been there already, to Q
20: **end if**
21: **end while**
22: **return** $\bar{x}_{\mathcal{V}}$
23: **end procedure**

In Table B.1 we present parameters applied in the procedure.

symbol	domain	value	description
threshold	$\mathbb{R}_+$	0.06	when all absolute values of coordinates of the force acting on a node are less than this value, the node is considered as stabilised
step_size	$\mathbb{R}_+$	0.125	a rate of a net force applied to stabilise a node in one step
max_counter	$\mathbb{Z}_{\geq 1}$	10	a maximal number of steps stabilising one node in a row

Table B.1: Table presents parameters applied in Algorithm 1 along with their domain, value used in test simulations, and description.

Step 10, where a node q is moved in the direction of the force acting on it, is an equivalent of a gradient descent modification (see C.2). In this step, each $i^{th}, i \in \{1, \ldots, 3\}$ coordinate of the node is decreased about the value proportional to the partial derivative of the Hamiltonian (see 2.12) with respect to the coordinate $x_q^{(i)}$ at the current configuration point $\bar{x}$:

$$x_q^{(i)} \leftarrow x_q^{(i)} - \eta \cdot \frac{\partial \mathcal{H}}{\partial x_q^{(i)}}(\bar{x}_{\mathcal{V}}), \tag{B.1}$$

where η is a small positive learning constant.

After calculating a net force for a given node q (see step 9), it is always dislocated in the direction of the force (see step 10). Even if the net force satisfies the stop condition defined by *threshold*. It is made since this step has low time cost and it predominantly enhances the node stabilisation. So, as a result, the vertex q is always dislocated at least once in the repeat loop 7.

It is vital that the larger value of the parameter *step_size* is, the bigger probability of a node oscillations. So, we have to keep this parameter sufficiently small. On the other hand, too small value of the parameter *step_size*, elongates the process significantly.

B.2 Parametric learning algorithm

The adaptation of parameters $k[e]$ and $\ell_0[e]$ is made with application of the gradient descent Algorithm 2 (see Section 2.3). In Table B.2 we present parameters applied in the algorithm.

symbol	domain	value	description
lear_rate	$\mathbb{R}_+$	$0.25 \cdot \lvert\mathcal{V}_{\text{obs}}\rvert$	size of a gradient descent step
max_slide	$\mathbb{R}_+$	1.3	a maximal acceptable ratio of the error $\Phi^{(i)}$ increment in one gradient descent step
k	$\mathbb{R}_+$	40	an initial value of k
δ	$\mathbb{R}_+$	0.2	a length of an interval applied to estimate difference quotient

Table B.2: Table presents parameters applied in Algorithm 2 along with their domain, value used in test simulations, and description.

The idea and the description of the step 5 introducing a noise are described in Section 2.5. The gradient $\nabla\Phi^{(i)}$ can be calculated explicitly given equation (2.12) and using a second order approximation of the Hamiltonian $\mathcal{H}$ at equilibrium $G[\bar{x}^0_{\mathcal{V}_{\text{movable}}}; \bar{y}^{(i)}_{\mathcal{V}_{\text{con}}}]$. However, this turns out to be quite inefficient as it requires inversion of large matrices. On the other hand, since the equilibrium $G[\bar{x}^0_{\mathcal{V}_{\text{movable}}}; \bar{y}^{(i)}_{\mathcal{V}_{\text{con}}}]$ has to be found anyway, by applying the dynamics (2.13), it is quite easy to approximate the gradient by directly examining the displacements of the equilibrium under small perturbations of parameters $k[e], \ell_0[e],\ e \in \mathcal{E}$. This is the option we have chosen. More precisely, in order to approximate partial derivatives of $\Phi^{(i)}$ with respect to $k[e]$ and $\ell_0[e],\ e \in \mathcal{E}$, we apply difference quotients:

$$\frac{\partial\Phi}{\partial k[e]} \simeq \frac{\Phi[(k[e]+\delta, \ell_0[e])] - \Phi[(k[e], \ell_0[e])]}{\delta},$$

$$\frac{\partial\Phi}{\partial \ell_0[e]} \simeq \frac{\Phi[(k[e], \ell_0[e]+\delta)] - \Phi[(k[e], \ell_0[e])]}{\delta}.$$

Algorithm 2 Parametric Learning Algorithm

procedure ParametricAdaptation($(E^{(i)})_{i=1}^{N}, \mathcal{G}$)
  while the stop condition is not satisfied **do**
    for each subsequent example $E^{(i)} = (\bar{y}^{(i)}_{\mathcal{V}_{\text{con}}}, \bar{y}^{(i)}_{\mathcal{V}_{\text{obs}}})$ cyclically **do**
      if learning with a noise factor **then**
        add a noise into the position of each control node $v \in \mathcal{V}_{\text{con}}$
        assign to $\bar{x}_{\mathcal{V}} \leftarrow$ Relaxation($\mathcal{G}$)
      end if
      set the positions of control nodes $\bar{x}_{\mathcal{V}_{\text{con}}}$ in accordance with $\bar{y}^{(i)}_{\mathcal{V}_{\text{con}}}$
      assign to $\bar{x}_{\mathcal{V}} \leftarrow$ Relaxation($\mathcal{G}$)
      calculate $\Phi^{(i)}$ and set $\Phi^{(i)}_{before} \leftarrow \Phi^{(i)}$
      calculate $\nabla\Phi^{(i)}(\bar{p}) \leftarrow$ DifferenceQuotient($\mathcal{G}, E^{(i)}, \Phi^{(i)}_{before}$)
      set $\bar{p} \leftarrow \bar{p} - lear\_rate \cdot \nabla\Phi^{(i)}(\bar{p})$
        ▷ $\bar{p}$ is a vecor of all parameters $p \in P : \bigcup_{e\in\mathcal{E}}\{k[e] \cup \ell_0[e]\}$
      assign to $\bar{x}_{\mathcal{V}} \leftarrow$ Relaxation($\mathcal{G}$)
      calculate $\Phi^{(i)}$ and set $\Phi^{(i)}_{after} \leftarrow \Phi^{(i)}$
      if $\Phi^{(i)}_{after} > max\_slide \cdot \Phi^{(i)}_{before}$ **then**
        revert the changes done in the step 12
      end if
    end for
    approximate the value of the error function $\Phi \leftarrow \frac{1}{N}\sum_{i=1}^{N}\Phi^{(i)}$
      ▷ for i^{th} training example $\Phi^{(i)} \leftarrow \Phi^{(i)}_{after}$ if the change
      ▷ of the parameters was accepted, otherwise, $\Phi^{(i)} \leftarrow \Phi^{(i)}_{before}$
    if it is the first epoch or Φ is the smallest up to now **then**
      store $k[e]$, $\ell_0[e]$, $e \in \mathcal{E}$ and $(G[\bar{x}^0_{\mathcal{V}_{\text{movable}}}; \bar{y}^{(i)}_{\mathcal{V}_{\text{con}}}])_{i=0}^{N}$
    end if
  end while
  return stored parameters $k[e]$, $\ell_0[e]$, $e \in \mathcal{E}$ and $(G[\bar{x}^0_{\mathcal{V}_{\text{movable}}}; \bar{y}^{(i)}_{\mathcal{V}_{\text{con}}}])_{i=0}^{N}$
end procedure

For a given training example $E^{(i)}$ and the spring system staying in the equilibrium $G[\bar{x}^0_{\mathcal{V}_{\text{movable}}}; \bar{y}^{(i)}_{\mathcal{V}_{\text{con}}}]$ with the value of the error function assigned to $\Phi^{(i)}_{before}$, the Algorithm 3 is used in order to determine the gradient $\nabla\Phi^{(i)}$.

Algorithm 3 Approximation of partial derivatives of $\Phi^{(i)}$ with respect to $k[e]$ and $\ell_0[e]$, $e \in \mathcal{E}$

1: **procedure** DIFFERENCEQUOTIENT($\mathcal{G}, E^{(i)}, \Phi^{(i)}_{before}$)
2: store $\bar{x}^{tmp}_{\mathcal{V}} \leftarrow \bar{x}_{\mathcal{V}}$
3: **for each** parameter p in the set $p \in P = \bigcup_{e\in\mathcal{E}}\{k[e] \cup \ell_0[e]\}$ **do**
4: $p \leftarrow p + \delta$
▷ δ is a small positive constant
5: assign to $\bar{x}_{\mathcal{V}} \leftarrow$ RELAXATION($\mathcal{G}$)
6: calculate $\Phi^{(i)}$ and set $\Phi^{(i)}_{after} \leftarrow \Phi^{(i)}$
7: $\dfrac{\partial\Phi^{(i)}}{\partial p} \leftarrow \Phi^{(i)}_{after} - \Phi^{(i)}_{before}$
8: $p \leftarrow p - \delta$
▷ withdraw the change made in the step 4
9: $\bar{x}_{\mathcal{V}} \leftarrow \bar{x}^{tmp}_{\mathcal{V}}$
▷ withdraw the change made in the step 5
10: **end for**
11: **return** $\nabla\Phi^{(i)}$
12: **end procedure**

B.3 Stop condition

A stop condition, which has to be satisfied in order to terminate execution of the parametric learning Algorithm 2, can be defined in different manners:

1. The stop condition is satisfied, when the error Φ (see 2.15) obtains value less than the predefined threshold. This condition, may never be satisfied or can significantly elongate the time of simulation, so it is not recommended.

2. We count the number of epochs (the number of runs of the loop while in line 2). The stop condition is satisfied, when the predefined number is obtained.

3. We count the number of epochs without improvement of the error Φ. The stop condition is satisfied, when the predefined number is obtained.

In our simulations we applied the third option. This choice guarantees us that the learning procedure converge and the time of processing is not elongated by useless epochs. For simulations in Chapters 3 and 4 we stop learning after 200 and 150 epochs without improvement, respectively.

B.4 Numerical precision of the calculation of the learning algorithm

The quality of adaptation of a spring system achieved by the training algorithm 2, depends on the values of parameters used by it and by algorithms which execute more precise tasks i.e. relaxation, approximation of partial derivatives of the error function. There are a few parameters which have an especially big effect on precision of calculation. These are:

1. $lear\_rate$ in the gradient descent algorithm 2,

2. *step_size* and *threshold* in the relaxation algorithm 1,

3. and δ, the value by which an argument (an equilibrium length or an elastic constant) of the error function Φ is increased to approximate its difference quotient 3.

The smaller values of these parameters are, the larger precision of the calculation is. But we have to remember that, when these parameters are decreasing also pace of adaptation is slowing down. Furthermore, we are limited by representation of floating point numbers in a computer architecture. We use parameters values, which could seem too large. It appears that for very small values of the mentioned parameters, simulations quickly get stuck in shallow local minima of the error function Φ. Another approach is to decrease values of the parameters in time, for example when for the significant number of epochs of the loop while 2, the parametric learning Algorithm 2 has not improved its adaptation (the value of the parameter *obtained_min* has not decreased). But our tests show that the Algorithm 2 on average obtains similar adaptation with and without descending parameters. So, we choose option with constant parameters, for which simulations are faster.

In the parametric learning Algorithm 2, when a spring system is adapted to the i^{th} training example, we check if the value of $\Phi^{(i)}$ does not increase significantly. The value of the parameter *max_slide*, applied to prevent the error from bursting, is equal to 1.3 (we approve worsening of the error function $\Phi^{(i)}$ maximally about 30 percent). This modification enables the learning process to get out of shallow local minima. Again, in order to save simulation time, we do not check how modification done during adaptation of the current training example impacts on $\Phi^{(j)}$ for the remaining examples. Consequently, it may happen that the algorithm approves a change with the value of the error function Φ considerably larger than it was before modification of the spring parameters. Figures 4.2 and 4.3 in Subsection 4.4.1 plot

the value of the error function Φ for exemplary learning simulations against the number of epochs of the parametric learning algorithm. There repeatedly the value of the error function Φ increases. These slides are momentary, thus, we ignore them.

For values of applied parameters, it can happen that for some spring parameters, values of approximated partial derivatives of the error function are very large. As a consequence, the relaxation Algorithm 1 significantly elongates its duration after decreasing spring parameters about gradient $\nabla\Phi^{(i)}$. Also, adaptation of the spring system becomes significantly worse and as a result, recently applied modification of all spring parameters, is rejected. We eliminate this problem by introducing a limitation on the maximal absolute value of each coordinate of the gradient.

B.5 Algorithm generating graph topology

Let us present the pseudocode of the algorithm generating graph topology. In Table B.3 we present parameters applied in the procedure.

symbol	domain	value	description
c	$\mathbb{R}^3$	$(0,0,0)$	the center and radius of a ball, whose surface contains the positions of $E^{(1)} = (\bar{y}^{(1)}_{\mathcal{V}_{\text{con}}}, \bar{y}^{(1)}_{\mathcal{V}_{\text{obs}}})$, these are common constants for this procedure and for the procedure 5
r	$\mathbb{R}_+$	200	
c_{aux}	$\mathbb{N}_0$	-	a number of auxiliary nodes in a graph $\mathcal{G}$
c_{fixed}	$\mathbb{N}_0$	2	a number of immobilised nodes in a graph $\mathcal{G}$
c_{edge}	$\mathbb{N}_{\geq 3}$	-	a coefficient for which $\lvert\mathcal{E}\rvert \approx c_{edge} \cdot \lvert\mathcal{V}\rvert$

Table B.3: Table presents parameters applied in the Procedure 4 along with their domain, value used in test simulations, and description. Values of parameters are given, if they are common for all tests in this dissertation.

Algorithm 4 Generating graph topology procedure

procedure TOPOLOGYGENERATION($E^{(1)} = (\bar{y}^{(1)}_{\mathcal{V}_{\text{con}}}, \bar{y}^{(1)}_{\mathcal{V}_{\text{obs}}})$)
  create a graph $\mathcal{G}$ with empty sets $\mathcal{V}_*$, $\mathcal{V}_{\text{fixed}}$, $\mathcal{V}_{\text{con}}$, $\mathcal{V}_{\text{obs}}$ and $\mathcal{E}$
  define an abstract ball with the radius r and centered at c
  for each $k \in \{1, \ldots, 4\}$ **do**
    randomly pick a node v uniformly distributed in the ball B
    link v with all nodes already added to the set $\mathcal{V}_*$
    add the node v to the set $\mathcal{V}_*$
  end for
  for each $k \in \{1, \ldots, c_{aux} + c_{\text{fixed}} - 4\}$ **do**
    randomly pick a node v uniformly distributed in the ball
    connect v with $min(|\mathcal{V}_*|, c_{edge})$ nearest nodes from $\mathcal{V}_*$
    add v to the set $\mathcal{V}_*$
  end for
  for each $k \in \{1, \ldots, c_{\text{fixed}}\}$ **do**
    fix a random auxiliary node v and move it from $\mathcal{V}_*$ to $\mathcal{V}_{\text{fixed}}$
  end for
  for each $\bar{y}^{(1)}_v$ in $\bar{y}^{(1)}_{\mathcal{V}_{\text{obs}}}$ **do**
    create a node v and assign $\bar{x}_v \leftarrow \bar{y}^{(1)}_v$
    connect v with $min(|\mathcal{V}|, c_{edge})$ nearest nodes from the set $\mathcal{V}$
    add v to the set $\mathcal{V}_{\text{obs}}$
  end for
  for each $\bar{y}^{(1)}_v$ in $\bar{y}^{(1)}_{\mathcal{V}_{\text{con}}}$ **do**
    create a node v and assign $\bar{x}_v \leftarrow \bar{y}^{(1)}_v$
    connect v with $min(|\mathcal{V}_{\text{movable}}|, c_{edge})$ nearest nodes from $\mathcal{V}_{\text{movable}}$
    add v to the set $\mathcal{V}_{\text{con}}$
  end for
  return the obtained structure $\mathcal{G} = (\mathcal{V}_* \cup \mathcal{V}_{\text{fixed}} \cup \mathcal{V}_{\text{con}} \cup \mathcal{V}_{\text{obs}}, \mathcal{E})$
end procedure

To obtain a rigid graph it is required that $c_{edge} \geq 3$. In order to eliminate the rigid motions of the graph $\mathcal{G}$, it has to be satisfied $|\mathcal{V}_{frozen}| = c_{\text{fixed}} + |\mathcal{V}_{\text{con}}| \geq 3$, see Appendix section A.

B.6 Algorithm for generation of training examples

Let us present the pseudocode of the procedure generating a set of training examples $(E^{(i)})_{i=1}^{N}$, for a predefined N, $|\mathcal{V}_{\text{con}}|$ and $|\mathcal{V}_{\text{obs}}|$. In Table B.4 we present parameters applied in the procedure.

symbol	domain	value	description
c	$\mathbb{R}^3$	$(0,0,0)$	the center and radius of a ball, whose surface contains the positions of $E^{(1)} = (\bar{y}^{(1)}_{\mathcal{V}_{\text{con}}}, \bar{y}^{(1)}_{\mathcal{V}_{\text{obs}}})$
r	$\mathbb{R}_+$	200	
θ_1	$(0,r) \in \mathbb{R}$	40	the maximal possible mean value of the distribution from which is sampled distance between $\bar{y}^{(1)}_v$ and $\bar{y}^{(i)}_v$ for $v \in \mathcal{V}_{\text{con}} \cup \mathcal{V}_{\text{obs}}$ for a given $i \in \{2,\ldots,N\}$
θ_2	$(0,r) \in \mathbb{R}$	40	the standard deviation of the distribution applied to sample coordinates $\bar{y}^{(i)}_v$ for $v \in \mathcal{V}_{\text{con}} \cup \mathcal{V}_{\text{obs}}$ for a given $i \in \{2,\ldots,N\}$

Table B.4: Table presents parameters applied in the Procedure 5 along with their domain, value used in test simulations, and description.

Algorithm 5 Generation of a set of training examples

procedure TRAININGEXAMPLEGENERATION($N, |\mathcal{V}_{\text{con}}|, |\mathcal{V}_{\text{obs}}|$)
  define a ball with the radius $r \in \mathbb{R}_+$ and centered at the point $c \in \mathbb{R}^3$
  partition the ball into two hemiballs: B_1 and B_2
  randomly pick uniformly distributed $|\mathcal{V}_{\text{con}}|$ points in the surface of B_1 and assign them to $\bar{y}^{(1)}_{\mathcal{V}_{\text{con}}}$
  randomly pick uniformly distributed $|\mathcal{V}_{\text{obs}}|$ points in the surface of B_2 and assign them to $\bar{y}^{(1)}_{\mathcal{V}_{\text{obs}}}$
  for each $i \in \{2, \ldots, N\}$ **do**
    randomly pick $l \sim \mathcal{U}(0, \theta_1)$
      $\triangleright$ θ_1 is a constant of the procedure
    randomly pick a vector $w \in \mathbb{R}^3$ and scale it to the length l
    for each $\bar{y}^{(i)}_v$ in $E^{(i)}$ **do**
      randomly pick a vector $\psi \sim (\mathcal{N}(0, \theta_2^2), \mathcal{N}(0, \theta_2^2), \mathcal{N}(0, \theta_2^2))$
        $\triangleright$ θ_2 is a constant of the procedure
      assign $\bar{y}^{(i)}_v \leftarrow \bar{y}^{(1)}_v + w + \psi$
    end for
  end for
  return $(E^{(i)})_{i=1}^N$
end procedure

B.7 A noise factor

Here we present the pseudocode of the Algorithm 6, which introduces a noise to the parametric learning algorithm. In Table B.5 we present parameters applied in the procedure.

Algorithm 6 Perturbation of positions of control nodes

procedure ADDNOISE($\mathcal{G}, (E^{(i)})_{i=1}^{N}$)
  for each control vertex $v \in \mathcal{V}_{\text{con}}$ **do**
    $[w_1, \ldots, w_N] \leftarrow$ sample coordinates from $\mathcal{U}_{(0,1)}$
    $s \leftarrow \sum_{i=1}^{N} w_i$
    for each $i \in \{1, \ldots, N\}$ **do**
      $w_i \leftarrow \frac{w_i}{s}$
    end for
    $m \leftarrow \sum_{i=1}^{N} w_i \cdot \bar{y}_v^{(i)}$
    randomly pick one point $\bar{x}'_v$ uniformly distributed in the ball centred at the point m, and with the radius equal to the algorithm parameter *noise_radius*
    change the position of the node v by ascribing $\bar{x}_v \leftarrow \bar{x}'_v$
  end for
end procedure

symbol	domain	description
noise_radius	$\mathbb{R}_+$	a parameter defining magnitude of a noise factor

Table B.5: Table presents parameter applied in the Procedure 6 along with its domain and its description.

B.8 Noise robustness

Here we present the pseudocode of the algorithm applied to examine noise robustness 7. The procedure input data are a trained system $\mathcal{G}$, its training examples and configurations of equilibria adapted for all of them — $(G^{(i)}_{patern})_{i=1}^{N}$. The value of a parameter L used in simulations is equal to 60.

Algorithm 7 Continuous transition between a pair of training examples

procedure INPUTNODESTRANSITION($\mathcal{G}, (E^{(i)})_{i=1}^{N}, (G^{(i)}_{patern})_{i=1}^{N}$)
  choose randomly two various training examples i and j
    ▷ optionally examples are predefined
  assign $\bar{x}_{\mathcal{V}} \leftarrow G^{(i)}_{patern}$
  for each control vertex $v \in \mathcal{V}_{\text{con}}$ **do**
    choose randomly a semicircle whose ends are attached to points $\bar{y}^{(i)}_v$ and $\bar{y}^{(j)}_v$ (optionally: take a straight line joining $\bar{y}^{(i)}_v$ and $\bar{y}^{(j)}_v$)
  end for
  for $l \leftarrow 1$ to L **do**
    for each control vertex $v \in \mathcal{V}_{\text{con}}$ **do**
      move $v \in \mathcal{V}_{\text{con}}$ along the curve determined for it by $\frac{1}{L}$ part of this curve in the direction of $\bar{y}^{(j)}_v$
    end for
    assign $\bar{x}_{\mathcal{V}} \leftarrow$ RELAXATION($\mathcal{G}$)
  end for
  $G^{(j)}_{obtained} \leftarrow \bar{x}_{\mathcal{V}}$
  $\Psi \leftarrow \frac{1}{|\mathcal{V}_{\text{movable}}|} \sum_{v \in \mathcal{V}_{\text{movable}}} \text{dist}([G^{(j)}_{patern}]_v, [G^{(j)}_{obtained}]_v)$
    ▷ Ψ is an average distance between positions of movable
    ▷ nodes in $G^{(j)}_{patern}$ and $G^{(j)}_{obtained}$
  return Ψ
end procedure

Appendix C

Backpropagation algorithm

There exist many similarities between the parametric learning and backpropagation algorithms. We enumerate them in Subsection 2.3.3. The backpropagation procedure is used to train *a multilayer perceptron* (MLP), which belongs to the family of artificial neural networks. We present their fundamental aspects in Section C.1. A gradient descent algorithm, which is an optimisation method applied by the backpropagation and parametric learning algorithms, we describe in C.2. Definition and functionality of a perceptron, which is the smallest learning unit of MLP, can be found in Section C.3. Finally, in Section C.4 we define the error function for a problem solved by MLP and by minimising it by the gradient descent scheme we derive the backpropagation algorithm.

C.1 Artificial neural networks

Artificial neural networks (ANNs) are models, which process information. They are inspired by biological nervous systems, such as the brain. The structure of the neural network is usually represented by a directed graph. Depending on the type of the neural network the edges can be bi-directed

or one-directed. The nodes of the network are usually identified with biological neurons and are called *processing elements*, *units* or *simple neurons*. The edges/connections of the artificial neural network are identified with *synapses* of neurons. The units connected by an edge can exchange information through this connection in the direction pointed by it. Each edge has assigned a numerical value called a *weight*, which scales the strength of the transmitted signal. Each unit retrieves signals from connections attached to it and directed into it, processes them and sends to other neurons through synapses. Some types of the neurons can store the processed information.

ANNs are applied to find solutions of problems that are hard to solve using ordinary programming methods e.g. pattern recognition or data classification. In order to solve a given problem, the neural network has to be *trained* through a *learning process*. There are three major learning paradigms. These are *supervised learning*, *unsupervised learning* and *reinforcement learning*. We would like to focus on supervised learning since this paradigm is relevant in the context of this dissertation. In this method of learning we are given a set of training examples. Each example consists of an input vector $x \in \mathbb{R}^I$ and an output vector $t \in \mathbb{R}^K$.

During the learning process the weights of the neural system are modified in such a way, that the neural network maps inputs x onto their outputs t in the best possible way. In order to employ the network to this transformation, we have to assign values from a given input vector to selected units in the artificial neural network. These units are called *input* ones. Next, the values from the control nodes are propagated through network synapses and eventually a numerical solution of the problem is returned through selected neurons called *output*. This usually requires an acyclic graph (DAG). The quality of the obtained mapping of the training examples is evaluated with application of an *error function* and depends on such factors like the structure of the neural network or complexity of the training set. So, the aim of

the learning process is to modify the weights in such a way that the sum of errors of all training examples obtains the lowest possible value. The neural system is adapted to training examples in a sequential way.

In this dissertation we discuss two neural network models trained with application of supervised learning. The first one is a *sigmoidal perceptron* and the second one is a *multilayer perceptron*. In fact, the first model is also the multilayer perceptron confined to one neuron. The learning process of both models is based on a gradient descent algorithm [10], so we start our discussion from describing this method.

C.2 Gradient descent algorithm

Gradient descent is an optimisation algorithm (GDA), applied to approximate a local minimum of a multivariate and differentiable function $f : \mathbb{R}^d \to \mathbb{R}$. More precisely we are given an argument $a^{(0)} \in \mathbb{R}^d$ and the goal of the algorithm is to find the minimum of an attractor, in which $f(a^{(0)})$ is located. The function f has to be continuous and differentiable. The gradient descent algorithm works in an iterative manner. Namely, during each i^{th} iteration, a point $a^{(i)}$ is calculated by taking a step proportional to the negative value of the gradient of the function f at the point $a^{(i-1)}$. The pseudocode of the algorithm is as follows:

1. Start at point $a^{(0)}$.

2. Perform:
$$a^{(i)} = a^{(i-1)} - \eta \cdot \nabla f(a^{(i-1)}),$$
where η is a small positive *learning constant*.

3. If a stop condition is met, return $a^{(i)}$, otherwise go back to 2.

A stop condition is satisfied, when each coordinate of $\nabla f(a^{(i-1)})$ is smaller than α (α is a small positive constant). It means that the point $a^{(i)}$ calculated in the last iteration is very close to the minimum, and it is an approximation of the minimum point.

The quality of the solution depends on the value of α. The smaller this value is, the closer to the sought minimum is the returned point.

The dynamics of GDA pushes a point a to the local minimum of its attractor. Because $a^{(0)}$ is a random point, it is worth to restart the algorithm many times and choose the minimum, for which the function f has the lowest value.

C.3 Sigmoidal perceptron

A simple perceptron is one of the most basic neural network models consisting of a single neural cell. A simple perceptron was developed by Frank Rosenblatt in 1958 [33]. Its definition is as follows:

Definition C.3.1. *A simple perceptron is a system consisting of n inputs $x_1, \ldots, x_n$ ($x_i \in \mathbb{R}$), $n+1$ weights $w_0, w_1, \ldots, w_n$ ($w_i \in \mathbb{R}$) associated with the inputs $x_0 = +1, x_1, \ldots, x_n$ and an activation or response function $f : \mathbb{R} \to \mathbb{R}$. Given the input vector $\bar{x} = [x_1, \ldots, x_n]$, the perceptron returns a response equal to:*

$$Out(\bar{x}) = f(\sum_{i=0}^{n} x_i w_i). \tag{C.1}$$

The input unit $x_0 = 1$ is called *a bias*, and in some cases it can significantly extend the learning ability of the perceptron. The function Out can have various forms. Commonly used are perceptrons with a sigmoidal activation function. The form of this function is as follows:

$$f(x) = \sigma(x) = \frac{1}{1 + e^{-\beta \cdot x}},$$

where $\beta \in (0, +\infty)$ is a parameter of the function. The plot of the sigmoidal function is presented in Figure C.1. The sigmoidal function returns values in the range $(0, 1)$.

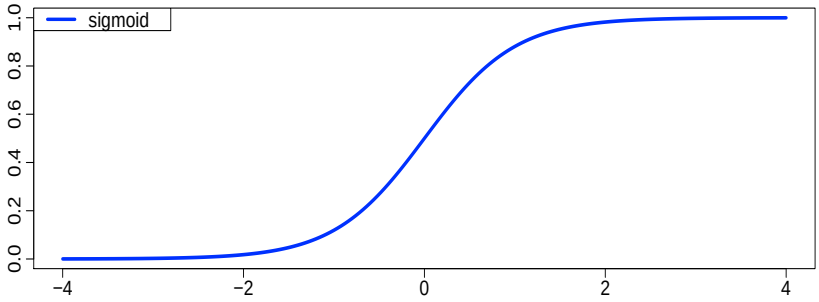


Figure C.1: The sigmoidal function.

During the learning process the derivative of this function is applied. It is easy to show that, the formula for the derivative of the sigmoidal function is as follows:

$$\sigma'(s) = \sigma(s)(1 - \sigma(s)).$$

The plot of $\sigma'(s)$ is depicted in Figure C.2.

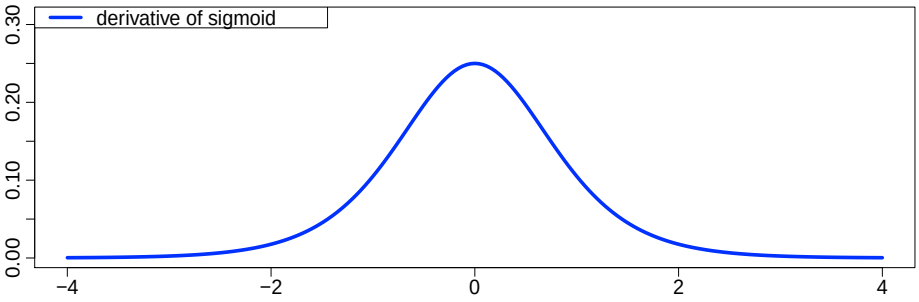


Figure C.2: The derivative of the sigmoidal function.

C.4 Backpropagation algorithm

A directed acyclic and topology-sorted graph of perceptrons is referred to as a *feed forward perceptron network*. A special type of such network is a *multilayer perceptron* (MLP). This type of network is able to approximate many complex functions. A user does not have to know or assume any form of dependencies in mapped models. It is a convenient tool for dealing with many forms of forecasting, classification, and automatic driven processes.

In the multilayer perceptron, all perceptrons are partitioned into sequential, disjoint layers. Neurons in the layer $L + 1^{st}$ as inputs get outputs from neurons in the previous layer L^{th}. It is prohibited to connect neurons in the same layer and neurons which are not in the sequential layer i.e. from L^{th} to $L + 2^{th}$, from L^{th} to $L + 3^{th}$ etc. Moreover, each unit in the L^{th} layer is connected with each unit in the $L + 1^{st}$ layer. We distinguish three types of layers in such network:

- the first/input layer, which is composed by input units,
- the last layer, which is composed by output units,
- all remaining layers between the input and output ones are called *hidden layers* (units in these layers are called *hidden units*).

In order to map the input vector x onto the output vector t with application of the multilayer perceptron, we assign values from a given input vector to respective neurons in the input layer and we propagate introduced signals forward through all hidden layers and finally through the output layer. Units in the $L+1^{st}$ layer can calculate their activations and propagate them further, if and only if they have obtained signals from all units in the previous L^{th} layer. After the propagation of the input signals through the whole network, the output layer returns the final mapped value.

The structure of the network is predefined before we proceed to the learning process. Namely, the number of hidden layers and number of neurons in each layer is known. For the sake of discussion simplicity, a network with one hidden layer is employed. Its input layer consists of I input units and additionally the bias, the hidden layer consists of J units and the bias, finally, the output layer consists of K units. The structure of this network is presented in Figure C.3.

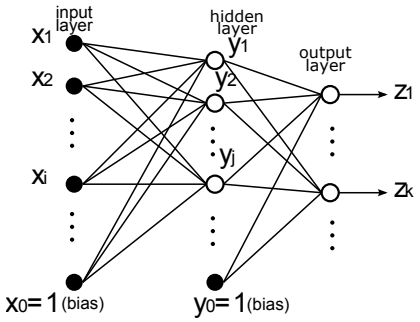


Figure C.3: A multilayer perceptron with one hidden layer.

Now, we introduce *forward* dynamics with application of appropriate notation, which will be helpful in the further part of this section. To set up the process, proper values are assigned to the neurons in the input layer and the signals are propagated through edges to the hidden neurons. Next, for each $j^{th}, j = \{1 \dots J\}$ neuron we calculate the weighted sum of obtained impulses, which is denoted by a_j:

$$a_j = w_j^t x.$$

Sequentially, for each j^{th} neuron is calculated the value of an activation function denoted by y_j:

$$y_j = f(a_j).$$

The activation function f has to be continuous and differentiable. Usually, it is represented by sigmoid. The activation returned by each neuron is sent

through synapses to units in the next layer. In Figure C.4 we depict edges connecting the input layer with the j^{th} neuron in the hidden layer.

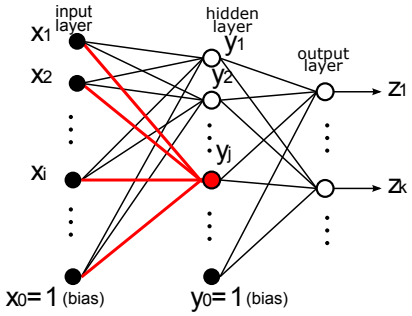


Figure C.4: Synapses connecting the input layer with the j^{th} neuron in the hidden layer are marked in red.

Next, analogically, activations for the hidden units are calculated and these signals are propagated to the units in the output layer:

$$b_k = w_k^t y,$$

$$z_k = f(b_k).$$

In Figure C.5 we depict edges connecting units in the hidden layer with the k^{th} neuron in the output layer.

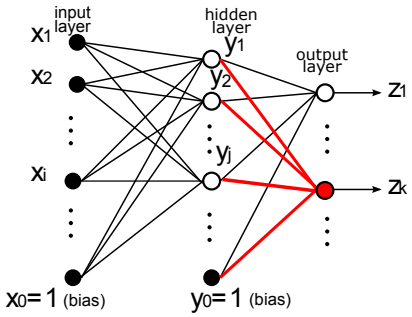


Figure C.5: Synapses connecting units in the hidden layer with the k^{th} neuron in the output layer are marked in red.

A learning algorithm for multilayer perceptron is based on the gradient descent algorithm and is called a *backpropagation algorithm* [39]. We assume that we are given a set of N training examples $((x^{(1)}, t^{(1)}), \ldots, (x^{(N)}, t^{(N)}))$. Our aim is to find values of weights, such that if the network as an input gets $x^{(n)} = (1, x_1^{(n)}, \ldots, x_I^{(n)})$, it returns a vector $z^{(n)} = (z_1^{(n)}, \ldots, z_K^{(n)})$, which is as close as possible to the expected output vector $t^{(n)} = (t_1^{(n)}, \ldots, t_K^{(n)})$, for all $n \in \{1, \ldots, N\}$. The quality of the adaptation of the neural system to all training examples is evaluated by the following squared error function:

$$ERROR(w) = \frac{1}{2} \sum_{n=1}^{N} \sum_{k=1}^{K} (z_k^{(n)} - t_k^{(n)})^2.$$

The arguments of this function are identified with weights of the network since z depends on them. So, we employ the gradient descent algorithm to find weights which minimise the error. What is vital that the $ERROR$ function is continuous and differentiable.

Let us denote the error function for one training example by $E^{(n)}, n \in \{1, \ldots, N\}$. So:

$$E^{(n)} = \frac{1}{2} \sum_{k=1}^{K} (z_k^{(n)} - t_k^{(n)})^2,$$

$$ERROR(w) = \sum_{n=1}^{N} E^{(n)},$$

$$E_k^{(n)} = \frac{1}{2} (z_k^{(n)} - t_k^{(n)})^2,$$

$$E^{(n)} = \sum_{k=1}^{K} E_k^{(n)}.$$

Additionally, the error function for one training example $E^{(n)}$ is split into the sum of expressions $E_k^{(n)}$, where $E_k^{(n)}$ is the squared error function for the n^{th} example of the k^{th} output of the network. In order to minimise the mean error $ERROR$, we minimise sequentially the error $E^{(n)}$ for all examples $n \in \{1, \ldots, N\}$. Namely, we make:

1. Start at random point $w^{(0)}$.

2. For all weights $g = 1 \dots d$ make:

$$w_g^{(m)} = w_g^{(m-1)} - \eta \frac{\partial E^{(n)}}{\partial w_g}(w^{(m-1)}),$$

   where η is a small positive *learning constant*.

3. If the stop condition is met, return $w^{(m)}$, otherwise go back to 2.

Initially, in step 2, the weights connecting the output layer with the hidden layer are modified. Next, the weights connecting the hidden layer with the input layer are altered. During calculation of output, we propagate input signals form the first layer to the last layer, whereas during weights modification we propagate the error $E^{(n)}$ from the last layer to the first layer. This is the reason of calling the discussed method *the backpropagation algorithm*.

During one step of adaptation of the network to $(x^{(n)}, t^{(n)})$, the weights are moved in the direction opposite to the gradient of $E^{(n)}$, but the formula for this vector of partial derivatives is not so clear and now we would like to derive it. In Figure C.6 we depict the weight w_{kj} joining the k^{th} unit in the output layer with the j^{th} unit in the hidden layer.

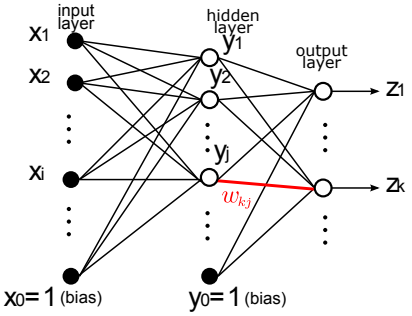


Figure C.6: A synapse connecting the k^{th} neuron in the output layer with the j^{th} unit in the hidden layer is marked in red.

First, we find formula for all such weights w_{kj} connecting the output layer

with the hidden layer. Since now, for notational simplicity we omit an index denoting the number of a training example in presented formulas (e.g. E means the same as $E^{(n)}$).

Since it is know, that:

$$z_k = f(b_k),$$

$$b_k = w_k^t y,$$

$$E = \frac{1}{2}\sum_{h=1}^{K}(z_h - t_h)^2 = \frac{1}{2}\sum_{h=1}^{K}(f(b_h) - t_h)^2,$$

it can be shown that:

$$\frac{\partial E}{\partial w_{kj}} = \frac{\partial}{\partial w_{kj}}\frac{1}{2}\sum_{h=1}^{K}(f(b_h) - t_h)^2 \frac{\partial}{\partial w_{kj}}\frac{1}{2}(f(b_k) - t_k)^2,$$

$$\frac{\partial E_k}{\partial w_{kj}} = \frac{\partial}{\partial w_{kj}}\frac{1}{2}(f(b_k) - t_k)^2 = \frac{\partial E_k}{\partial b_k}\frac{\partial b_k}{\partial w_{kj}} = \delta_k y_j.$$

In the previous formula, for notational convenience, we denoted $\dfrac{\partial E_k}{\partial b_k}$ by δ_k. It is easy to show that:

$$\frac{\partial E_k}{\partial b_k} = \delta_k = (z_k - t_k)f'(b_k).$$

In turn:

$$\frac{\partial b_k}{\partial w_{kj}} = \frac{\partial}{\partial w_{kj}}(w_{k1}y_1 + w_{k2}y_2 + \cdots + w_{kj}y_j + \cdots + w_{kJ}y_J) = y_j.$$

So:

$$\frac{\partial E}{\partial w_{kj}} = \frac{\partial E_k}{\partial b_k}\frac{\partial b_k}{\partial w_{kj}} = \delta_k y_j.$$

Now, we find partial derivatives for weights w_{ji} connecting the hidden layer with the input layer. In Figure C.7 we depict the weight w_{ji} connecting the j^{th} neuron in the hidden layer with the i^{th} unit in the input layer.

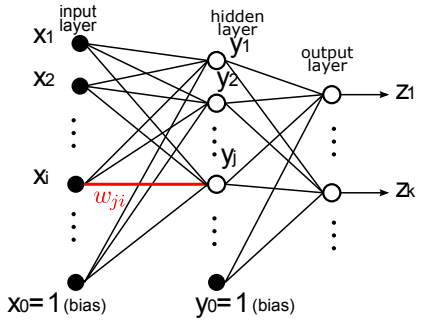


Figure C.7: A synapse connecting the j^{th} neuron in the hidden layer with the i^{th} unit in the input layer is marked in red.

Since E depends on a_j and a_j depends on w_{ji}, from the chain rule we can get:

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}}.$$

First, we simplify the formula for the partial derivative of a_j with respect to w_{ji}:

$$\frac{\partial a_j}{\partial w_{ji}} = \frac{\partial}{\partial w_{ji}}(w_{j1}x_1 + w_{j2}x_2 + \cdots + w_{ji}x_i + \cdots + w_{jI}x_I) = x_i.$$

Next, we simplify the formula for the partial derivative of E with respect to a_j. Since we know that:

$$a_j = w_j^t x,$$

$$b_k = w_k^t y,$$

$$y_j = f(a_j),$$

it can be shown that:

$$\frac{\partial E}{\partial a_j} = \frac{\partial}{\partial a_j}\frac{1}{2}\sum_{h=1}^{K}(f(b_h) - t_h)^2 = \sum_{h=1}^{K}\frac{\partial}{\partial a_j}\frac{1}{2}(f(b_h) - t_h)^2 = \sum_{h=1}^{K}\frac{\partial E_h}{\partial a_j} =$$

$$= \sum_{h=1}^{K}\frac{\partial E_h}{\partial b_h}\frac{\partial b_h}{\partial a_j} = \sum_{h=1}^{K}\delta_h\frac{\partial b_h}{\partial a_j}.$$

Now, we calculate the partial derivative of b_h with respect to a_j:

$$\frac{\partial b_h}{\partial a_j} = \frac{\partial}{\partial a_j}(w_{h1}y_1 + w_{h2}y_2 + \cdots + w_{hj}y_j + \cdots + w_{hJ}y_J) =$$

$$\frac{\partial}{\partial a_j}(w_{h1}f(a_1) + w_{h2}f(a_2) + \cdots + w_{hj}f(a_j) + \cdots + w_{hJ}f(a_J)) =$$

$$\frac{\partial w_{hj}f(a_j)}{\partial a_j} = \frac{w_{hj}\partial f(a_j)}{\partial a_j} = w_{hj}f'(a_j),$$

what we apply to obtain:

$$\sum_{h=1}^{K} \delta_h w_{hj} f'(a_j) = (\sum_{h=1}^{K} \delta_h w_{hj}) f'(a_j).$$

We denote the obtained formula for the partial derivative of E with respect to a_j by δ_j. So, finally we get:

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}} = \delta_j x_i.$$

Having derived formulas for partial derivatives of the error function E with respect to weights, we can introduce backpropagation algorithm:

1. Randomly pick initial weights.

2. Sequentially for each training example (x, t) do:

   (a) Perform forward propagation of the input signals through the network, for each unit remember its input sum and its output. So, for the hidden nodes we calculate:

   $$a_j = w_j^t x,$$

   $$a_j = w_j^t x,$$

   in turn, for the output nodes:

   $$b_k = w_k^t y,$$

   $$z_k = f(b_k).$$

(b) Perform backward propagation of the error E through the network, for each unit calculate δ value. First we calculate δ values for units in the output layer:

$$\delta_k = (z_k - t_k) f'(b_k).$$

Next, for the units in the hidden layer:

$$\delta_j = (\sum_{k=1}^{K} \delta_k w_{kj}) f'(a_j).$$

Keeping in mind that for the output units:

$$f'(b_k) = z_k(1 - z_k),$$

in turn, for the hidden units:

$$f'(a_j) = y_j(1 - y_j),$$

if f is the sigmoidal function.

(c) Update the weights. For the output layer apply the formula:

$$w_{kj}^{(m+1)} = w_{kj}^{(m)} - \eta \delta_k y_j.$$

Equivalently, for the hidden layer:

$$w_{ji}^{(m+1)} = w_{ji}^{(m)} - \eta \delta_j x_i.$$

Here, η is a small positive learning constant e.g. $\eta = 0.001$.

3. If the error $ERROR$ is still decreasing, go back to 2.

The multilayer perceptron trained by the backpropagation algorithm can have more than one hidden layer. For many learning problems it is advisable, to apply at least two hidden layers. For each unit in hidden layers, δ is calculated in the same way. One must keep in mind that before values of δ for the units in the L^{th} layer are calculated, these values have to be calculated for units in the $L + 1^{st}$ layer. δ for a given unit in the L^{th} layer, which is hidden, depends on δ-s from the $L + 1^{st}$ layer.

Acknowledgments

The author would like to mention prof. Tomasz Schreiber (1975–2010), a brilliant mathematician and computer scientist, professor at the Faculty of Mathematics and Computer Science, Nicolaus Copernicus University. He had the greatest influence on the subject of this work and direction of scientific interests of the author.

The author acknowledges the help of supervisor — prof. Jacek Miękisz for trust and patience.

The author would like to thank Jarosław Piersa for many discussions concerning scientific interests and motivation to finish the dissertation.

Bibliography

[1] T. G. Abbot, *Generalizations of Kempes Universality Theorem*, MSc thesis, MIT (2008) 81

[2] B. D. O. Anderson, B. Fidan, J. M. Hendrickx, C. Yu: *Rigidity and Persistence for Ensuring Shape Maintenance of Multiagent Meta Formations*, Asian Journal of control, 10, Issue 2 (special issue on Collective Behaviour and Control of Multi-Agent Systems), 131–143, (2008) 81

[3] L. Asimov, B. Roth, *The Rigidity of Graphs*, Trans. Amer. Math. Soc. 245, 279–289, (1978) 79

[4] A. R. Atilgan, S. R. Durell, R. L. Jernigan, M. C. Demirel, O. Keskin, I. Bahar, *Anisotropy of fluctuation dynamics of proteins with an elastic network model*, Biophysical journal, 80(1), 505–515, DOI: 10.1016/S0006-3495(01)76033-X 11, 12, 40

[5] H. M. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. N. Bhat, H. Weissig, I. N. Shindyalov, P. E. Bourne, *The Protein Data Bank*, Nucleic Acids Research 28, 235–242, (2000) 7, 9, 39

[6] R. Connelly, W. Whiteley, *Second-Order Rigidity and Prestress Stability for Tensegrity Frameworks*, SIAM Journal of Discrete Mathematics 9, 453–491, (1996) 12, 20, 79

[7] R. Connelly, *Rigidity and energy*, Inventiones Mathematicae 66, 11–33, (1982) 12, 20

[8] R. Connelly, *Generic global rigidity*, Discrete Comp. Geometry 33, 549–563, (2005) 81

[9] A. L. Cauchy, *Sur les polygones et les polyèdres* XVIe Cahier IX, 87–89, (1813) 79

[10] A. L. Cauchy, *Méthode générale pour la r'esolution des systèms déquations simultanées*, Comp. Rend. Sci. Paris, 25, 46–89, (1847) 103

[11] M. Czoków, T. Schreiber, *Adaptive Spring Systems for Shape Programming*, ICAISC 2010, Part II, LNAI 6114, 420–427, (2010) 7, 10, 19

[12] M. Czoków, T. Schreiber, *Structure Searching for Adaptive Spring Networks for Shape Programming in 3D*, ICAISC 2012, PART II, LNCS 7268, 207–215, (2012) 7, 10, 30

[13] M. Czoków, J. Miękisz, *Influence of a Topology of a Spring Network on its Ability to Learn Mechanical Behaviour*, PPAM 2013, PART I, LNCS 8384, 412–422, (2014) 7, 10

[14] G. M. Crippen, T.F. Havel, *Distance Geometry and Molecular Conformation*, Wiley, New York, (1988) 11, 12

[15] A. A. Gusev, *Finite Element Mapping for Spring Network Representations of the Mechanics of Solids*, Phys. Rev. Lett. 93, 034302, (2004) 11

[16] E. Granato, S. C. Ying, *Dynamical transitions and sliding friction in the two-dimensional Frenkel-Kontorova model*, Phys. Rev. B 59, 5154, (1999) 11

[17] J. Graver, B. Servatius, H. Servatius, *Combinatorial Rigidity*, Graduate Studies in Math., AMS, (1993) 82

[18] *W. Greiner, Classical Mechanics: Systems of Particles and Hamiltonian Dynamics*, Classical theoretical physics, Springer, (2010) 85

[19] B. Jackson, *A necessary condition for generic rigidity of bar-and-joint frameworks in d-space*, arXiv:1104.4415, (2011) 81

[20] M. Kellomäki, J. Aström, J. Timonen *Rigidity and Dynamics of Random Spring Networks*, Phys. Rev. Lett. 77, 2730, (1996) 11

[21] A. Kilian, J. Ochsendorf, *Particle–Spring Systems for Structural Form Finding*, Journal of the International Association for Shell and Spatial Structures: IASS, 46, (2005) 11

[22] S. Kmiecik, D. Gront, M. Kolinski, L. Wieteska, A. E. Dawid, A. Kolinski: *Coarse-Grained Protein Models and Their Applications*, Chemical Reviews 2016 116 (14), 7898–7936, (2016) 40

[23] J. Kovacs, P. Chacón, R. Abagyan: *Predictions of protein flexibility: first-order measures* Proteins, 56(4), 661–668, (2004) 40

[24] G. Laman, *On graphs and rigidity of plane skeletal structures*, J. Eng. Mathematics, 4, 331–340, (1970) 82

[25] P. Maragakis, M. Karplus, *Large amplitude conformational change in proteins explored with a plastic network model: adenylate kinase*, J. Mol. Biol., 2005, 352, 807–822, (2005) 40

[26] J. A. Mccammon, B. R. Gelin, M Karplus, *Dynamics of folded proteins*, Nature, 267, 585–590, (1977) 40

[27] O. Miyashita, *Nonlinear elasticity, proteinquakes, and the energy landscapes of functional transitions in proteins*, Proc. Natl. Acad. Sci. USA, 100, 12570–12575, (2003) 40

[28] K. I. Okazaki, N. Koga, S. Takada, J. N. Onuchic, P. G. Wolynes, *Multiple-basin energy landscapes for large-amplitude conformational motions of proteins: Structure-based molecular dynamics simulations*, Proc. Natl. Acad. Sci. USA, 103, 11844–11849, (2006) 40

[29] R. Olfati-Saber, R. M. Murray, *Graph Rigidity and Distributed Formation Stabilization of Multi-Vehicle Systems*, Proc. of the 41st IEEE Conf. on Decision and Control, Las Vegas, Nevada, (2002) 12, 79, 81

[30] L. Orellana, M. Rueda, C. Ferrer-Costa, J. R. López-Blanco, P. Chacón, M. Orozco, *Approaching Elastic Network Models to Molecular Dynamics Flexibility*, J. Chem. Theory Comput. 2010, 6, 2910–2923, (2010) 40

[31] L. Orellana, *Large-Scale Conformational Changes and Protein Function: Breaking the in silico Barrier*, Frontiers in molecular biosciences, (2019) 39

[32] M. Ostoja-Starzewski: *Lattice Models in Micromechanics*, Appl. Mech. Rev. 55, 35–60, (2002) 11

[33] F. Rosenblatt, *A probabilistic model for information storage and organization in the brain*, Psychological Review, 65(6), 386–408, (1958) 104

[34] M. Rueda, P. Chacón, M. Orozco, *Thorough validation of protein normal mode analysis: a comparative study with essential dynamics*, Structure, 15(5), 565–575, (2007) 41

[35] P. Sfriso, A. Hospital, A. Emperador, M. Orozco, *Exploration of conformational transition pathways from coarse-grained simulations*, Bioinformatics, 2013 15 Aug, 29(16), 1980–1986, (2013) 40, 41, 42, 43, 44

[36] F. Tama, Y. H. Sanejouand, *Conformational change of proteins arising from normal mode analysis*, Protein Engineering, 14, 1–6, (2001) 39

[37] V. Tozzini, *Coarse-grained models for proteins*, Current opinion in structural biology, 15, issue 2, 144–150, (2005) 40

[38] A. E. Varvitsiotis, *Algebraic and combinatorial techniques in rigidity theory*, MSc thesis, (2009), http://users.uoa.gr/ avarvits/MSc.pdf 81, 83

[39] P. Werbos, *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*, PhD thesis, Harvard University, (1974) 109

[40] W. Zheng, M. Tekpinar, *Analysis of protein conformational transitions using elastic network model*, Methods Mol Biol 2014, 1084, 159–172. 40