ASPERITY DYNAMICS: NUMERICAL MODELING OF SINGLE ASPERITY CONTACTS AT SOLID-SOLID INTERFACES

by

Arnfinn Mihle Paulsrud

Thesis

for the degree of

Master of Science

(Master i Fysikk, studieretning Computational physics)

Faculty of Mathematics and Natural Sciences
University of Oslo

March 2014

Contents

List of Figures

List of Tables

Part I

Introduction

Chapter 1

Introduction

1.1 The History of Friction

Throughout human history friction has had a great impact on mankind, and according to Dowson [14] one of the first encounters man had with friction was through the discovery of fire in the Old Stone Age. Though little is known about this prehistoric period it is believed that the early man controlled generation of fire by means of the percussion of flint stones and the friction of wood on wood, frictional heating.

Around 4,000 years ago, the Egyptians used their knowledge about friction and lubrication to move heavy objects. In Figure 1.1 we kan see a sketch of the statue of an Egyptian colossus, with some estimates of a coefficient of friction needed to move this heavy object with 172 men. As we can see the coefficient of friction is estimated to be 0.23, which is very close to coefficient of friction for lubricated wood. One of the interesting parts of this sketch is the coefficient of friction estimated in the lower right corner. As we can see the

The first recorded quantitative study of friction was conducted by Leonardo da Vinci. His motivation for studying friction was his concern about the role of friction in the performance of screw-jacks and gears. The difference between Leonardo and his predecessors was that he used the scientific method when he conducted his experiments. Leonardo measured the force of friction between objects on bouth horizontal and inclined surfaces. Some of the experiments he conducted is essentially the same as students learn in physics class today. In Figure 1.2 we can see some of his experimental setup. Through his experiments Leonardo recognized the difference between rolling and solid friction and the beneficial effect of lubricants. Perhaps two of his greatest observations during his experimentation was (i) that the friction made by the same weight will be of equal resistance at the beginning of its movement although the contact may be of different breadths. (ii) Friction produces double the amount of effort if the weight be doubled. These two observations is consistent with what we now know as the two first laws of friction, namely

1. The force of friction is directly proportional to the applied load.

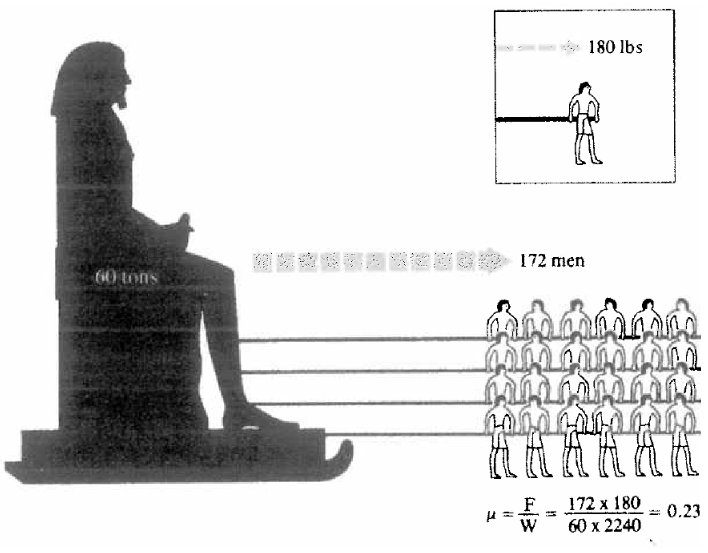
180 lbs
60 tons
172 men
$\mu = \frac{F}{W} = \frac{172 \times 180}{60 \times 2240} = 0.23$

Figure 1.1: These are some of Leonardo da Vinci's sketches of friction experiments. We can see different types of sliders, with different contact area and pulley systems to create experiments with constant velocities

2. The force of friction is independent of the apparent area of contact.

Leonardo also observed that the frictional resistance depended upon the nature of the surface in contact, and that bodies with smoother surfaces have smaller friction. He was also the first to introduce the consept of the coefficient of friction as the ratio of the force of friction to the normal load. He concluded that the coefficient of friction had a value of 0.25 for all materials. This was probably a good estimate at that time, given the types of materials and instruments of measurement.

At the end of the seventeenth century in France, Guillaume Amontons also tried to expand his knowledge about the field of friction. Like Leonardo, he also used experiments to gain a better knoowledge about friction. In figure 1.3 we can see Amontons experimental setup for friction experiments. The specimens tested were of copper, iron,lead and wood in various combinations. The test specimens like A-A and B-B were loaded together with various springs depicted by C-C-C, and the force required to overcome the friction force and initiate sliding was measured on the spring balance D.

1. That the resistance caused by rubbing only increases or diminishes in proportion to greater or lesser pressure (load) and not according to the greater or lesser extent of the surface.

2. That the resistance caused by rubbing is more or less the same for iron, lead and copper and wood in any combination if the surface is coated with pork fat.

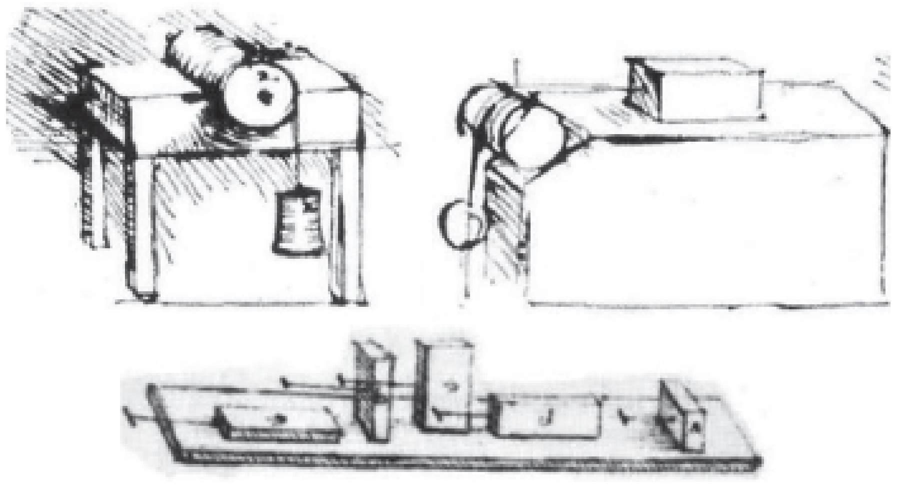

Figure 1.2: These are some of Leonardo da Vinci's sketches of friction experiments. We can see different types of sliders, with different contact area and pulley systems to create experiments with constant velocities

3. That this resistance is more or less equal to one-third of the pressure (load).

We can see that the first observation embodies the first and second laws of friction. One interesting thing about Guillaume Amontons experiments is that he actually used pork fat as a lubricant between the specimens. This means that he was actually studying the frictional characteristics of boundary lubrication.

The next very important step in the history of the study of friction, was the discoveries done by Charles Augustin Coulomb. Charles Augustin Coulomb set out to investigate the influence of four main factors upon friction:

1. The nature of the materials in contact and their surface coatings.

2. The extent of the surface area.

3. The normal pressure (load).

4. The length of time that the surfaces remained in contact (time of repose).

Later he also studied the influence of ambient conditions like temperature, humidity and vacuum. He put in a great effort to explain that the to major hypotheses which had been introduced by earlier workers to explain friction related to asperity interactions, introduced by Amontons, and cohesion which was introduced by Desaguliers. He discovered that under dry conditions the friction between unlubricated wooden surfaces reached a constant value after periods of rest of one or two minutes and that the typical values where as follows:

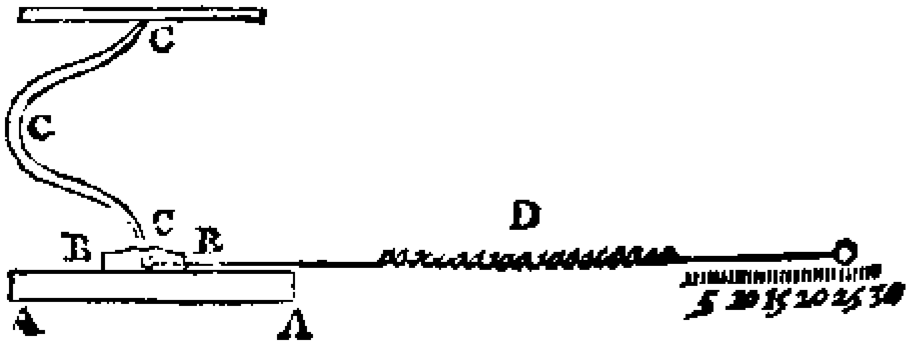

Figure 1.3: Amontons' sketch of his apparatus for friction experiments. **A-A** is the base. **B-B** is the slider. **C-C-C** is a spring that provides normal loading. **D** is a spring balance with scale for friction measuremens.

Table 1.1: Coefficient of friction recorded by Charles Augustin Coulomb

Materials	**Weight/Friction**
Oak sliding on oak	2.34
Elm sliding on elm	2.18
Pine sliding on pine	1.78
Oak sliding on pine	1.5

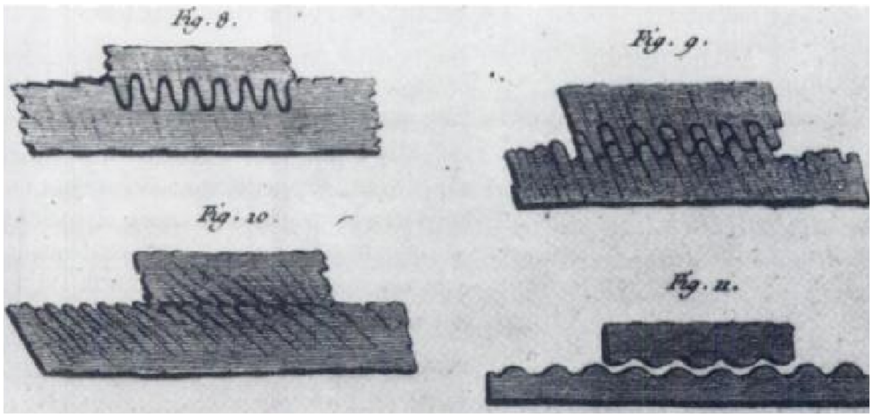

Figure 1.4: Here we can see how Coulomb imagined how rough surfaces interacted with each other.

By presenting such data he exposed his essentially pragmatic approach and his desire to provide usable data. The data in Table 1.1 was used by Coulomb to construct empirical equations relating the force of friction to this variable. Coulomb found that in most cases friction was almost proportional to load and independent of the size of the contacting surfaces. The foundations of his theory was summarized by four principal features of his experimental findings.

1. For wood sliding on wood under dry conditions the friction rises initially but soon reaches a maximum. Thereafter the force of friction is essentially proportional to the load.

2. For wood sliding on wood the force of friction is essentially proportional to the load at any speed, but kinetic friction is much lower than the static friction related to long periods of repose.

3. For metals sliding on metals without lubricant the force of friction is essentially proportional to load and there is no difference between static and kinetic friction.

4. For metals on wood under dry conditions the static friction rises very slowly with time of repose and might take four, five or even more days to reach its limit. With metal-on-metal the limit is reached almost immediately and with wood-on-wood it takes only one or two minutes. For wood-on-wood or metal-on-metal under dry conditions seed has very litle effect on kinetic friction, but in the case of wood-on-metal the kinetic friction increases with speed.

Coulomb concluded, based on these observations, that friction could come only from the meshing of asperities. In Figure 1.4 see how Coulomb imagined how rough surfaces interacted with each other.

Between 1850 and 1925 some new questions arose due to the development of the railway. During this period a good deal of discussion about the relationship between static and kinetic friction were made. Some considered the two coefficients to be quite different, with a sharp transition from one value to another as sliding commenced or ceased. The experimental evidence presented by several workers gradually confirmed that the friction of motion and the friction of rest could be represented by a continuous function. Experiments also revealed that the kinetic friction could be higher than the static friction given the right circumstances.

In 1929 Tomlinson assumed that both the normal load and the tangential force of friction where linearly related to the number of interacting atoms, but in the absence of more detailed knowledge of surface deformation abd intermolecular force he was unable to quantify the phenomenon.
Deryagin made a bold attempt to quantify some of the concepts outlined by Tomlinson in a statistical approach involving detailed representations of intermolecular forces and crystal structure. He derived a two-term expression for friction. (Details of the work have been questioned and debated, but there is little doubt that it was a valuable step forward from the tentative proposals put forward by Tomlinson).
Holm carried out a comprehensive study of the manner in which electricity was conducted from one solid to another in such devices as terminals, relays and circuit-breakers and by 1938 he was convinced that clean metal surfaces would deform plastically at asperity contacts and cold-weld. He concluded that one element of force of friction must be attributable to the sum of the shearing strengths of the asperity contacts.
Bowden and Tabor's approach was also based upon the recognition that surfaces in contact touched only at points of asperity interaction and that the very high stresses induced in such regions of small area would lead readily to local plastic deformation. The penetration of an asperity into the opposing surface could be likened to a miniature hardness test and the mean normal stress ,p, over the real areas of asperity contact, a, could be represented to all intents and purposes by the hardness,H, of the softer material. Likewise, if s represented the shear stress of the asperity junctions, the normal load, P, friction force, F, and the coefficient of friction, μ, could express with appealing simplicity by the relationships

$$P = aH \tag{1.1}$$

$$F = as \tag{1.2}$$

Combining the above equations gave an expression for the coefficient of friction

$$\mu = F/P = s/H \tag{1.3}$$

This expression for the coefficient of friction in terms of established mechanical properties of materials represented a great step forward in the theory of friction, although it is clearly incomplete. It neglects the detailed, more complex nature of asperity interactions and deformation and, of course, accounts only for the adhesive element of friction. The limitations are apparent when when it is recognized that, for metals,

$$s \approx 0.5\sigma_y \tag{1.4}$$

$$H \approx 3\sigma_y \tag{1.5}$$

Where σ_y is the yield stress in tension. All clean metals should thus exhibit a universal coefficient of friction of 1/6, which is qualitatively and satisfyingly consistent with the statements of Leonardo da Vinci and Amontons, but unfortunately unrepresentative of more sensitive experimental findings.
Ernst and Merchant In their study of the metal-cutting process they attempted to take account of surface roughness and the fact that the real area of contact between asperities might be inclined at an angle (theta) to the direction of overall sliding. The resulting expression for mu was similar to those of Bowden and Tabor, but involved additional terms in $\tan(\theta)$ on the right-hand side of the equation.
In the third approach to the nature of friction it was argued that a force would be required to move hard asperities through or even over another surface and that this micro-cutting motion represented the friction process. The idea had received its first serious recognition from Gumbel and Everling (1925) and it is still the subject of detailed studies in the field of plasticity theory. The idea was simple enough. If the sum of the projected areas of the indenting asperities perpendicular to the direction of sliding is a_0, and the mean stress resisting plastic deformation of the sifter material which is being cut is equal to the hardness ,H, the total force of friction $F = a_0H$. Likewise, if the applied load, W, is carried on the number of asperity contacts of real area ,a, $W = aH$. The coefficient of friction thus becomes

$$\mu = F/W = a_0/a \tag{1.6}$$

For conical asperities having sides sloping at a mean angle theta to the direction of sliding

$$\mu = a_0/a = 2/\pi \tan(\theta) \tag{1.7}$$

While for hemispherical asperities of radii, R, and small penetration

$$\mu = a_0/a = 4/3\theta/\pi \tag{1.8}$$

Since theta is small, typically 5-10 deg, it can be seen from the above equations that alternative specifications of asperity geometry have little effect upon the magnitude of the friction force attributable to plastic deformation. If it is assumed that both molecular (adhesive) and deformation (ploughing) actions are effective, then the resulting expression for mu is simply a combination of Eq. 1.3 and either 1.7 or 1.8. Thus

$$\mu = s/H + 2/\pi \tan(\theta) \tag{1.9}$$

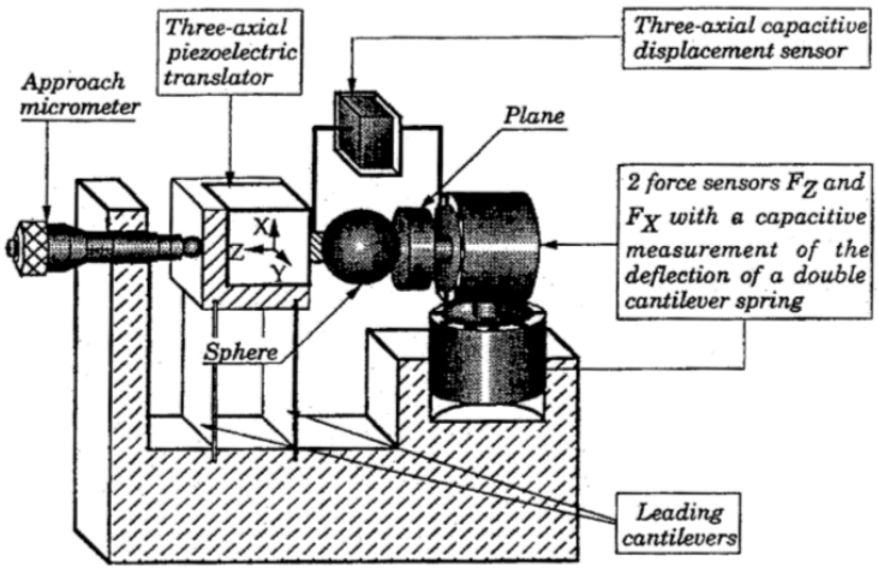
Three-axial piezoelectric translator
Approach micrometer
Three-axial capacitive displacement sensor
Plane
2 force sensors F_Z and F_X with a capacitive measurement of the deflection of a double cantilever spring
X
Z
Y
Sphere
Leading cantilevers

Figure 1.5: Three-axial surface force apparatus (SFA) from the group of Georges in Lyon for measuring forces between a sphere and a plane. Source: NanoScience, Meyer and Overney et al., World Scientific

In a series of experiments carried out in Melbourne, Australia during the Second World War, Bowden et al. used sliding contacts of differing geometries to ascertain the relative importance of adhesion and ploughing. The sliders included a sphere, a circular section spade and a cylinder with its axis parallel to the direction of sliding. The finding demonstrated that adhesion played the major role in determining the friction between metals, and although this conclusion has to be treated with some caution and related to specific material combinations, it is still regarded as a valid observation.
Courthney-Pratt and Eisner (1957) drew attention to the mechanism of junction growth. An assumption was that the area of real contact was determined by the normal force alone, but when the combined normal and shear stresses are considered and a yield criterion introduced, it becomes clear that the real area of contact can increase many times before sliding occurs. The net result of this is that the force and hence the coefficient of friction increases considerably above the predictions of Eq.1.3. The concept of junction growth is one of the most exciting developments in the field of friction studies in the recent years. However, it remains clear that the continued application of plasticity theory of the friction problem will further enhance our understanding of the process.
On the experimental front, two forms of instrument have contributed enormously to recent progress.
The surface force apparatus (SFA), Figure 1.6, works by having two very smooth solids, such as cleaved mica, pressed together to enable atomic scale friction measurement. On the nano-scale, the force of friction was seen to be directly related to the real area of contact,

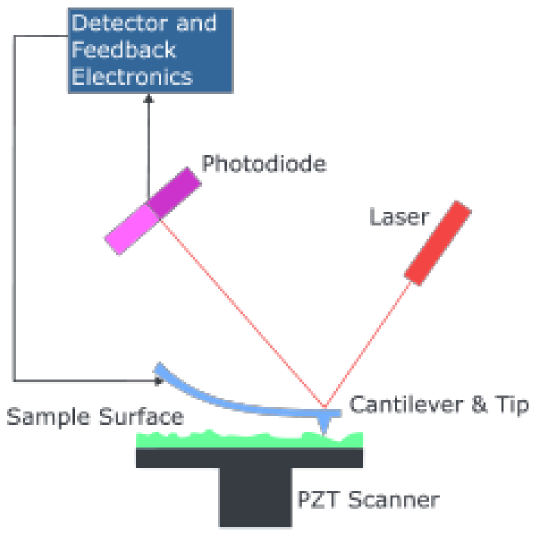

Figure 1.6: Block diagram of atomic force microscope using beam deflection detection. As the cantilever is displaced via its interaction with the surface, so too will the reflection of the laser beam be displaced on the surface of the photodiode. Picture taken from wikipedia [4]

but the concept that friction and wear were linked to adhesive junctions and the plucking out of material from one of the solids was not sustained. Friction did not correlate well with the cohesive strength of the solids and it was noted that wear-free friction could be measured. It was the irreversibility associated with the process of bringing atoms together and then separating them, rather than the force of cohesion itself, that correlated with friction. Tomlinson had proposed a link between friction and interacting atoms which was developed more fully in the 1970s. McClelland and Glosli (1992) then developed a simple model of friction based upon vibrations of atomic lattices in which the work done in overcoming friction was dissipated through vibrations (sound) and eventually as heat.
The atomic force microscope (AFM), Figure 1.6, has a very fine probe with a tip of radius in the range 10-100 nano-metre, which enable measurements to be made on single asperity contacts. The probe traverses over a surface and the inter-atomic forces between the probe, and the test surface can be measured to determine force components with pico-newton accuracy. At the atomic scale, the force of friction is no longer proportional to load, since friction depends upon the true area of contact. This in turn is determined on the atomic scale not only by the applied normal force, but also by adhesion. Even when the external applied force is zero, the contacting solids will "flatten" under the action of adhesive forces. The physics has been outlined by Israelachvili (1992), while the contact mechanics has been analyzed by Johnson et al. (1997).

1.2 Goals with this thesis

1. 1d model with Amontons-Coulomb with interfacial stiffness with side and uniform driving: Find sequence of precursors and load curve, plot precursor length as a function of driving force. Visualize stress curves (F_N, F_S, $\mu_s * F_N$ and $\mu_d * F_N$). Study, for example, the influence of the normal load tilt, the friction coefficients, the internal-to-interfacial stiffness ratio and the system resolution (number of blocks).

2. Add preconditioned stress to 1d model. For the cases of initial Poisson-effect as described in Amundsen et al, 2012 measure the precursor length as a function of driving force for both side-driven and uniformly driven systems (with non-uniform loading for uniform driving).

3. Start simulations from part 1 before the first precursor by instead unloading (changing F_N). Visualize the stress curves. Find sequence of precursors. Plot precursors as a function of F_T/F_N.

4. Apply method from part 3 to cases in part 2. Visualize the stress curves. Find the sequence of precursors. Plot precursor as a function of F_T/F_N Study unloading with double side driven model (driven from both sides, in opposite directions).

1.3 The structure of the thesis

In chapter two I'll go trough some of the basic theory of friction, and then I'll explain some of the friction models used today.
In chapter three I'll explain how a modern friction experiment is conducted and how the setup of the apparatus of the group at the University of California, Berkeley that work with unloading experiments is set up. In this chapter I'll also explain some of the constants that are used in this thesis and where they come from.
In chapter four I'll go through the numerical method that has been used in this thesis.
In chapter five I'll go through the one dimensional numerical friction model. Both side driven and top driven.
In chapter six I'll show the one dimensional unloading model, and some results from the numerical simulations.
In chapter seven I'll conclude the thesis, and take a look at what was accomplished and what the next steps regarding the unloading model might be.

Part II

Theory

Chapter 2

Friction Theory

Friction is a part of the scientific field called Tribology, which incorporate the study of lubricants, lubrication, friction, wear and bearings.
Like gravity, friction is one of the first consepts we learn about in physics. And like gravity, friction is a force that affect our lives all the time. Every time we get out of bed, friction is there to helps us to get grip between our feet and the floor. When we go for a drive, friction is there to help the car to get traction. Without friction life as we know it would be impossible.
When we first learn about friction, we are taught that friction is the force that tries to resist motion of solid surfaces. The simplest equation of friction, which states that the friction equals the normal force times some constant, is given as

$$F_f = \mu F_N, \tag{2.1}$$

where μ is the friction coefficient and F_N is the normal force. As we can see from the above equation, the friction force does not depend on the contact area. It is common to split the friction force into two categories, static friction and dynamic friction. Static friction is probably the form of first the friction force we learn about. This is the force that prevents motion. If we have a system as in Figure 2.1, the force, F_x, whould have to be greater than the friction force, F_f, for the box to move. The static friction force is given by the static friction coefficient, μ_s, and the equation for static friction is given by the equation

$$F_{fs} = \mu_s F_N. \tag{2.2}$$

As the driving force gets larger than the static friction force, the block begins to move. At that moment, when the driving force gets larger than the static friction force, the static friction force changes to a dynamic friction force. The dynamic friction force is given by the dynamic friction coefficient, μ_d, which is lower than the static friction coefficient [5] . Sice the dynamic friction coefficient is lower than the static friction coefficient, we need a smaller force to keep the block moving after it has reached the static friction threshold. The equation for dynamic friction is given by the equation

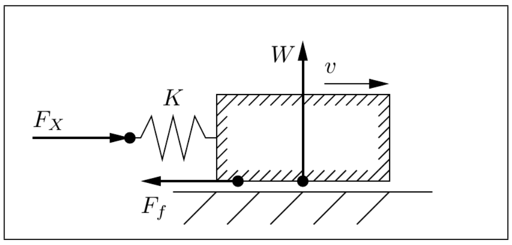

Figure 2.1: A block on a surface which has a driving force, F_x, a normalforce, W, and a friction force, F_f

$$F_{fd} = \mu_d F_N. \tag{2.3}$$

This description of friction is called Amontons-Coulomb friction.

It has also been shown through experiments that μ_s is not a constant number, it is actually a value that is slowly increasing and it depends on the so-called waiting time, t_w. The waiting time is the time between static friction, when we start to apply the driving force, and sliding, when we have dynamic friction. So the longer the waiting time, the larger the static friction coefficient gets. This effect is often called aging.
As we saw in the last chapter Amontons and Coulomb was two of the pioneers in the development of frictional framework we have today. From their experiments and research three friction laws has emerged, these are known as the Amontons-Coulomb law's of friction [5]:

1. No motion occurs as long as the driving force ,F_X , is smaller than a finite threshold $\mu_s F_N$.
2. The friction force is independent of the apparent area of contact.
3. When motion occurs, the friction force is also proportional to the normal load.

2.1 The Burridge-Knopoff model

In the late 1960s R. Burridge and L. Knopoff [11] was working on a mathematical one-dimensional model for describing earthquake faults. In this model Burridge and Knopoff uses a chain of connected masses (or particles)
Each mass is connected to its neighboring masses by coil springs and to a overlying rigid support by a flat spring. This rigid support moves horizontally with a velocity, V. The coil springs has a spring constant $\mu_1, \mu_2, \mu_3, ..., \mu_n$ and the flat springs has a spring constant $\lambda_1, \lambda_2, \lambda_3, ..., \lambda_n$. Burridge and Knopoff had also done some experiments with an actual connected spring model, and by setting $\lambda_1 = \mu_1 = \mu_2 = ... = \mu_n \neq 0$ and $\lambda_2 = \lambda_3 = ... = \lambda_n = 0$ they obtained what we now call a side driven model. They now had a model to describe the interaction between the masses. They now needed some description of the interaction between the masses and the surface and some way to combine these two

things. The model for friction that they chose included the effects of instability radiation and viscosity. These properties was baked into a function $F(v_i)$, where v_i is the the ith mass. They now combined the above into the equation of motion

$$m_i \frac{\partial^2 x_i}{\partial t^2} = T_i - T_{i-1} + T_i^* + F_i^* \qquad \text{for} i = 1, 2, ..., N \tag{2.4}$$

where $T_i = \mu_i(x_{i+1} - x_i)$, $T_i^* = -\lambda_i(x_i - Vt)$ and m_i is the mass of each particle (mass). By taking Eq.(2.4) a little bit further they came up with the energy equation

$$\frac{\partial}{\partial t}\{\sum_1^N \frac{1}{2} m_i (\frac{\partial x}{\partial t})^2 + \sum_1^{N-1} \frac{1}{2}(x_{i+1} - x_i)^2 + \sum_1^N \frac{1}{2}\lambda_i (Vt - x_i)^2\} \tag{2.5}$$

$$= \sum_1^N V\lambda_i(Vt - x_i) - \sum_1^N E_i(\frac{\partial x}{\partial t})^2 + \sum_1^N \frac{\partial x}{\partial t} F(x_i) \tag{2.6}$$

The terms in the equation above each describes a physical property of the system $\sum_1^N \frac{1}{2} m_i (\frac{\partial x}{\partial t})^2$ is the kinetic energy of the system. $\sum_1^{N-1} \frac{1}{2}(x_{i+1} - x_i)^2$ is the potential energy in the connecting springs. $\sum_1^N \frac{1}{2}\lambda_i(Vt - x_i)^2$ is the potential energy in the flat springs connecting each particle to the moving support. $\sum_1^N V\lambda_i(Vt - x_i)$ is the rate of doing work in moving the support against the flat springs and is of order V which can be made small. $\sum_1^N E_i(\frac{\partial x}{\partial t})^2$ is the power radiation along the semi-infinite strings and is positive. $-\sum_1^N \frac{\partial x}{\partial t} F(x_i)$ is the rate at which the work is done against friction and viscosity. It too is positive.

2.2 A simpler friction model

The friction model used by Burridge and Knopoff figure 2.1 is a very complicated friction model. In [17] S. Meagawa, A. Suzuki and K. Nakano uses a similar block model as Burridge and Knopoff, but with a friction model more similar with the static and dynamic friction model introduced earlier in this chapter. The model consists of a number of masses (or blocks) connected by coil springs. The system of connected masses are driven in a point P connected through a coil spring to the first mass and is driven with a constant velocity, V. This is quite similar to the Burridge-Knopoff model. The big difference comes when the frictional force is introduced. The friction force which act on each mass is given as

$$f_i = \begin{cases} f_s^{(i)}, & \text{when} \frac{dx}{dt} = 0 \\ -f_k^{(i)}, & \text{when} \frac{dx}{dt} > 0 \\ f_k^{(i)}, & \text{when} \frac{dx}{dt} < 0. \end{cases} \tag{2.7}$$

This description of the friction model gives us a quite simple model, and a pure Amonton-sâĂŞCoulomb friction model, that does not make so many assumptions as the Burridge-Knopoff model do. Meagawa et al. also introduces a non-uniform normal force to the friction model given as

$$w_i = \frac{F_Z}{N}\left(1 + \frac{2i - N - 1}{N - 1}\theta\right) \tag{2.8}$$

where the total normal load is given as $F_Z = \sum_{i=1}^{N} w_i$, the total number of masses is N and the non-uniformity of the normal force is given by $-1 \leq \theta \leq 1$. $\theta = 0$ gives us uniform normal force. The effect of this non-uniform normal force is

1. Higher load at the trailing edge relative to that at the leading edge ($F_{ZA} > F_{ZB}$) results in a smaller number of precursor events than under uniform loading conditions ($F_{ZA} = F_{ZB}$).
2. Non-uniform loading conditions with a higher load at the leading edge ($F_{ZA} < F_{ZB}$) lead to a larger number of precursor events relative to that in uniform loading conditions.
3. non-uniform loading additionally affects the increasing rate of the propagation length of the precursors.

2.3 Viscous damping

A problem with the spring-block model is the coil springs between the block (or masses). When a block goes from a state of static friction to a state of dynamic friction the block gains a high velocity. The effect of this velocity gain is a constant switching of energy between the coil spring to the right of the block and the coil spring to the left of the block. If nothing is done with this, the block will constantly change between negative and positive velocity In their master thesis, J. Trømborg [27], D.S Amundsen [2] introduces an additional feature to the spring-block model. This addition is the viscous damping which helps convert the some of the kinetic energy in the system into potential energy. This fixes the oscillating behavior of the blocks. The equation for the viscous damping is given as

$$F_i^{\eta} = \begin{cases} \eta(\dot{x}_2 - \dot{x}_1), & i = 1 \\ \eta(\dot{X}_{i+1} - 2\dot{x}_i + \dot{x}_{i-1}), & 2 \leq i \leq N-1 \\ \eta(\dot{u}_{i-1} - \dot{x}_N), & i = N \end{cases} \tag{2.9}$$

where $\dot{x}_i$ is the velocity of the block,and $\dot{X}_{i+1}$ and $\dot{X}_{i-1}$ is respectively the velocity of the left and right block. F_i^{η} is the force due to the velocity and η viscous damping.

Part III

Experiments

Chapter 3

Experiments

During the last decade there have been conducted experiments involving Poly(methyl methacrylate) (PMMA) blocks rather than the usual materials we know from the history of friction research where the more common material like copper, iron,lead or wood blocks that were used. At first glance this might seem a little bit strange because PMMA is not a very commonly used construction material. It's a very brittle material and is therefor not very suitable for withstanding large forces.

The strength of PMMA in the field of experimental friction studies is its transparency. With the development of sensors with high resolution, it is now possible to record what is happening in the transition between the slider and the base in experiments. This technology has made it possible to look at the asperities.

At the Reacah Institute of Physics in Jerusalem, J. Fineberg and his group have conducted experiment [7, 8, 23, 22, 24, 25] with sliding PMMA blocks. At Yokohama University in Japan, Maegawa, Suzuki and Nakano [17] has conducted some of the same experiments.

In this chapter we'll first take a look at some experimental setups for modern friction experiments and some of their results. And then we'll look at an experiments done by one of S.D. Glaser students at the University of California, Berkeley.

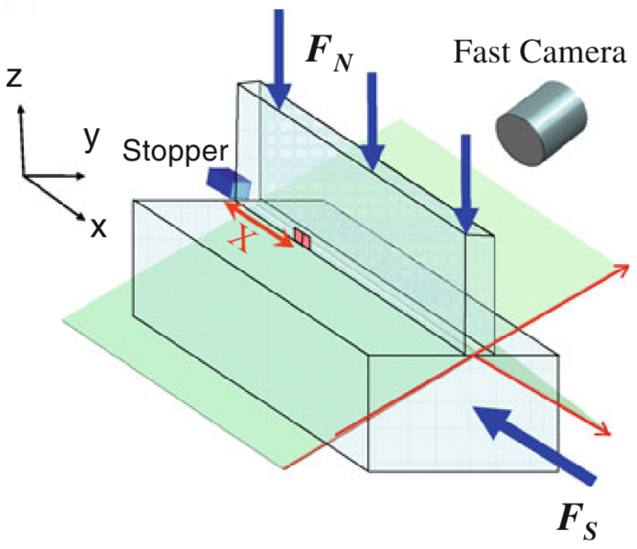

Figure 3.1: A schematic view of the experimental setup of Fineberg et al. [6]

3.1 Experiments

In the experiments that where conducted by Fineberg et al. the experimental setup consisted of two blocks of PMMA. In Figure 3.1 we can see a sketch of the two PMMA blocks. The upper block, called the slider, which in the experiments of Fineberg et al. [6] had the dimensions 200 mm x 6 mm x 100 mm. The bottom block, which is called the base had the dimensions 300 mm x 30 mm x 28 mm in the sliding (x), transverse (y) and loading (z) directions respectively. The contact surface of base and the slider where both treated to get a desired and consistent rough surface of approximately $1\mu m$. During the experiment the upper block exposed to a uniform normal loading, F_N. This force was monitored throughout the experiment via a load cell with a stiffness of 10^7 N/m. The shear force (driving force) F_S was applied to the bottom block. This block was mounted on a low friction linear stage, and its motion in the x direction was only constrained by the frictional force at the interface with the slider. At the trailing edge, $x = 0$, the slider was constrained by a stopper to prohibit motion in the x-direction. The shear force, F_S, was applied to the base via a load cell with a stiffness of 10^6 N/m. To detect slip events, an acoustic sensor was mounted on the trailing edge to detect slip events. When a slip event was detected, F_S was held for a pre-defined hold time. In Figure 3.2 we can see how the slip at a single point, $\delta(X, t)$, where measured.
To measure this, a laser beam was focused on a metallic grid glued to the side of the slider approximately 2 mm above the frictional interface, the faces in contact. The laser had a good enough resolution to measure a slip of $0.2\mu m$. The measured as shown in 3.1. Here we can see a laser light that is being directed through the frictional interface. If there is contact between the two blocks the laser light will be detected on the oposite site of the laser light source. If there is no contact between the two blocks the laser light will be

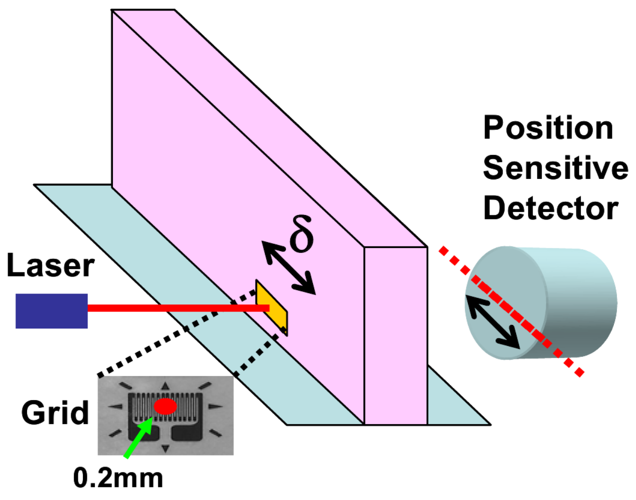

Figure 3.2: A sketch of the experimental setup by Ben-David et. al [8]. The pink colored block is the slider, and the cyan colored is the base. A laser is used to measure the displacement of the slider.

deflected by the difference in the refractive index of the PMMA and the air, and no light will be detected in that point.
In Figure 3.3 and Figure 3.4 we can see some of the results from the experiments done by Ben-David et. al. in Figure 3.3 **a)** we can see the how the normalized contact area between the slider and the base changes during the experiment. at $t = 0$ s, we see that the contact area is uniform throughout the whole frictional interface. As the experiment goes on the shear force increases, Figure 3.3 **b)** , and the contact area increases. At $t \approx 150$s we can see that there is a sudden change at the trailing edge, this indicates that there has been a precursor, and each precursor is initiated at the trailing edge. In Figure 3.4 we can see the slip, $\delta(X, t)$, as a function of time compared to the normalized contact area.
Here we can see four different phases of how the contact area, $A(X, t)$, and the slip, $\delta(X, t)$, is affected before, during and after front passage through location X. Phase I is the detachment phase, this phase is followed by rapid slip (phase II) which sharply transitions into slow slip (phase III). Although the contact area is reduced by $\approx 20\%$ during the detachment phase, it remains relatively constant during the ensuing slip phases.

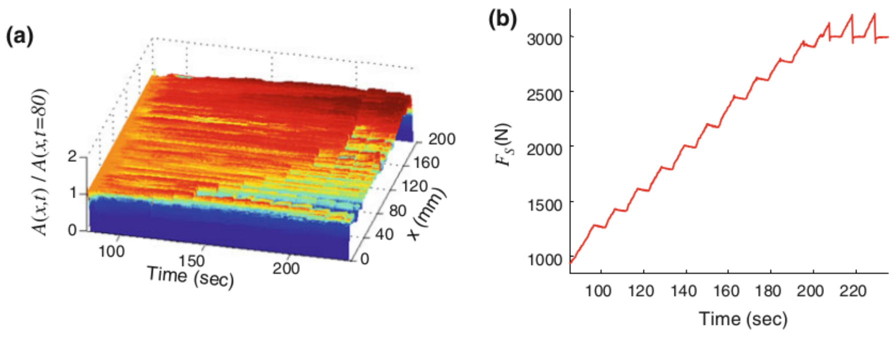

Figure 3.3: In **a)** we can see a measurement of the normalized contact area, $A(x, t)$, normalized by the value of $A(x, t = 80)$. Red colors indicate increased A, while blue color indicated reduction of A. In **b)** we can see how the shear force, F_S, changes during the experiment. From [6]

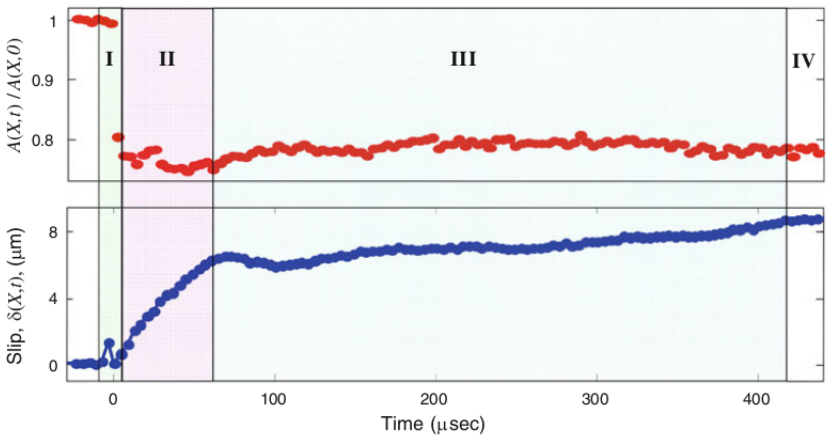

Figure 3.4: Here we can see the detachment and evolution of frictional slip

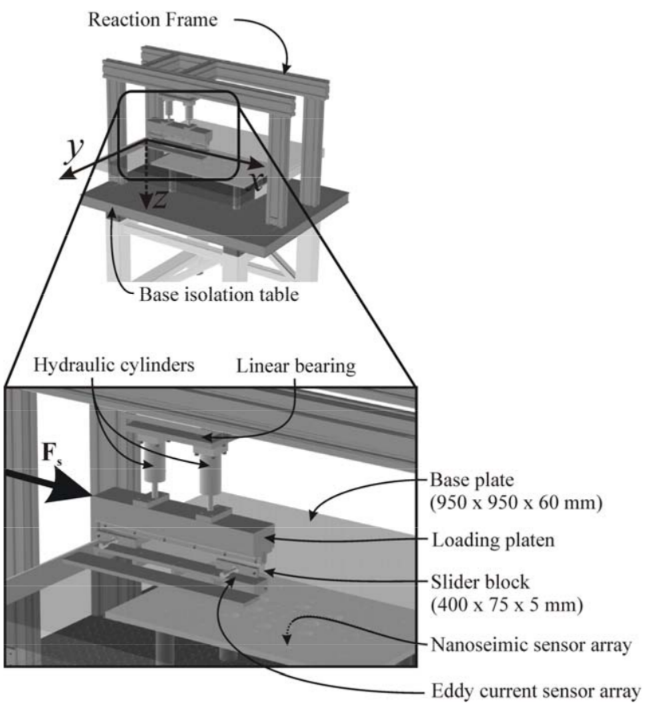

Figure 3.5: A sketch of the friction apparatus used for friction PMMA experiments at the University of California, Berkeley [26]

3.2 Unloading model

At the University of California, Berkeley P.A. Selvadurai [26] and S.D. Glaser [18] are working with seismic stress and sliding friction. In Figure 3.5 we can see the apparatus use by both Selvadurai and Glaser in their friction experiments.
Although the apparatus is used in different way in [26] and [18], Selvadurai is currently using it for unloading experiments. In Figure 3.6 we can see the sample block from Selvadurai's experiment. The samples are 1x40 cm PMMA blocks. The sliding surface is sandblasted to a given roughness, and the experiment is performed rough on rough with a base that has had the same sandblasting treatment. The sample is glued to the top holder, this is always at the same place, which is rigid compared to the sample. The holder has a grove that the slider fits into and it has been machined by computer to give very good precision for parallel top and bottom surface, straightness etc.
Normal load, Figure 3.7, is applied through the top holder. The PMMA is attached to a fixed frame and this frame is attached to two hydraulic pressure cylinders. The two hydraulic pressure cylinders is connected to a self-aligning joint (a ball) Through pressure control in the hydraulic fluid they give a normal force control.
Shear load, Figure 3.8, is applied to the top holder The motor, that apply the shear load, is attached after the normal force has been applied. Shear load can be displacement controlled or force controlled, depending on the situation.

3.3 Parameters of the model

In this thesis I am going to do some numerical simulations of some of the systems explained in the above experiments. To get the best possible results the parameters used in the numerical model should be as close to the experimental values. As most of the frictional sliding experiments are done with PMMA, I will also use PMMA in my simulations. According to wikipedia [20] PMMA has a Young's modulus in the range of $1.8 - 3.1$ GPa. In [17] Maegawa et al. used the value 2.5 GPa. In this thesis I'll also use this value. The rest of the values i Table 3.1 have been taken from [27], [2] and [17].

Table 3.1: Parameters used in the 1D model

Physical quantity	**Symbol**	**Value in simulation**
Total slider mass	M	0.012 kg
Young's modulus	E	2.5 GPa
Sample size in x-direction	L_x	100 mm
Sample size in y-direction	L_y	5 mm
Sample size in z-direction	L_z	20 mm
Static friction coefficient	μ_s	0.70
Dynamic friction coefficient	μ_d	0.45
Relative viscous damping	η	$\sqrt{0.01}\sqrt{km}$
Applied normal load	F_Z	400 N
Driving point velocity	V	0.1 mm/s
Driving spring constant, side driven model	K	0.8 MN/m
Driving spring constant, top driven model	K_n	K/N

Table 3.2: This table shows the parameters that have been used in this thesis. The parameters have been taken from [27], [2] and [17]

Some of the values will be changed to see how the change affects the behavior of the system. The relative viscous damping is 1/10 of what was used in [27], the reason for this is some strange behavior explained in Appendix A. The driving spring constant for the top driven model is given as the driving spring constant for side driven model divided by the number of blocks, N.

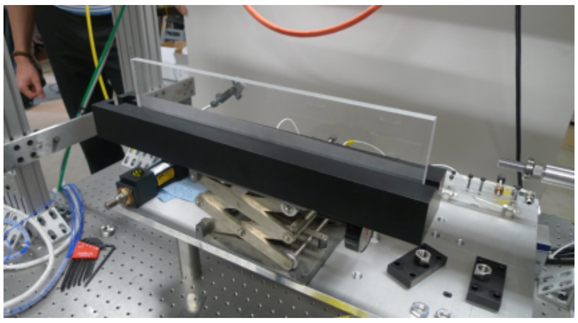

Figure 3.6: A picture of how the block is attached to the apparatus. Photographed by J.Trømborg

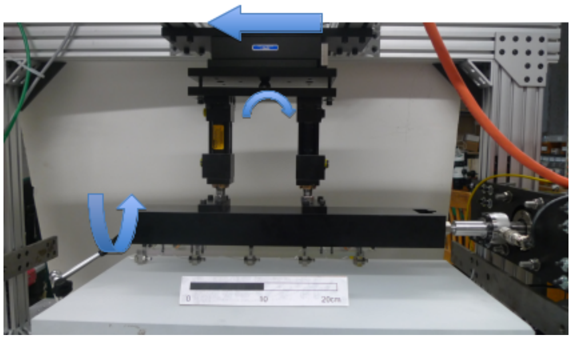

Figure 3.7: A picture of how the normal load is applied to the block. Photographed by J.Trømborg

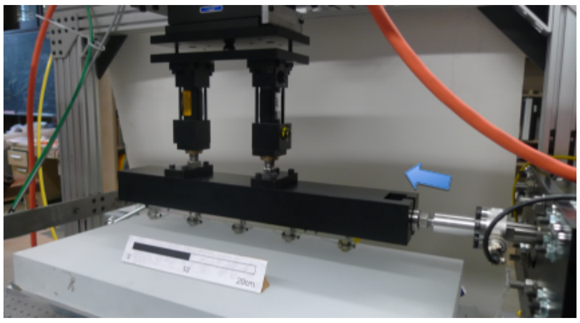

Figure 3.8: A picture of how the shear load is applied to the system. Photographed by J.Trømborg

Part IV

Numerical Methods and Ordinary Differential Equations

Chapter 4

Numerical Methods

Numerical methods has played a big part in modern physics and science, and in the comming years it'll probably play an even bigger role. One of the first courses we encounter when we start studying physics is some kind of course in mechanics. In this course we learn about Newton's law's and how to solve then analytically. One of the things the lecturer often forget to mention, is the limited number of problems that actually have an analytical solution. Fortunately we have found a way around this problem by using numerical methods.

Some of problems we use numerical methods to solve in physics and science is:

1. Ordinary differential equations (Euler method, Runge-Kutta method)
2. Integration (numerical integration, Monte Carlo [19])
3. Partial differential equations (finite difference, finite element)
4. Eigenvalue problems (finding eigenvalues and eigenvectors)
5. Simulating physical systems (molecular dynamics [1])

4.1 Ordinary differential equations

A great many applied problems involve rates, that is, derivatives. An equation containing derivatives is called a differential equation. If it contains partial derivatives it is called a partial differential equation, otherwise it is called an ordinary differential equation (ODE).

A good example of an ODE is Newton's second law of motion

$$\sum F(t) = ma(t), \tag{4.1}$$

where $\sum F(t)$ is the total force involved in the system, m is the mass of the system and $a(t)$ is the acceleration. It is possible to write the acceleration as derivatives $a(t) = dv(t)/dt =$

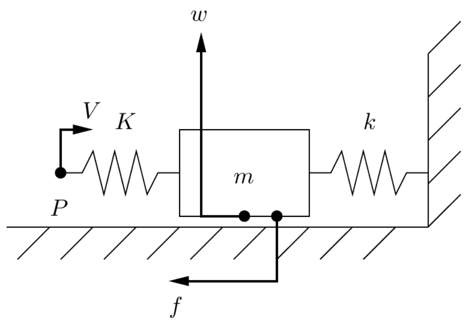

Figure 4.1: A one-block friction system. V is the velocity of point P, m is the mass of the system, K is the driving spring stiffness, f is the friction force and k is the spring stiffness between the block and the wall. The full solution of this system can be found in [2]

$d^2x(t)/dt^2$. If we now insert this into Newton's second law of motion we get a well known ODEs

$$\frac{d^2x(t)}{dt^2} = \frac{\sum F(t)}{m} \tag{4.2}$$

Depending on the complexity of $\sum F(t)$ we can either solve this analytically or numerically. The system shown in Figure 4.1 is one of the systems it is possible to solve analytically. The forces arising from the two springs in Figure 4.1 follows Hook's law, which is an elastic spring law on the form

$$F(t) = kx(t), \tag{4.3}$$

where k is the spring stiffness and $x(t)$ is the position from equilibrium. If we use this equation on the system in Figure 4.1 we get

$$\frac{d^2x(t)}{dt^2} = \frac{\sum F(t)}{m} + \frac{kx(t)}{m}, \tag{4.4}$$

which is a second-order differential equation. It is possible to rewrite this second-order differential equation into two first-order equation

$$\frac{dx(t)}{dt} = v(t) \tag{4.5}$$

$$\frac{dv(t)}{dt} = \frac{\sum F(t)}{m} + \frac{kx(t)}{m} \tag{4.6}$$

The reason for dividing second-order differential equation into two first-order equation is to make it easier to solve with numerical methods.

4.2 The numerical methods

When it comes to solving ordinary differential equations numerically, there are many different methods that we can use. The best known methods are probably the Euler method and the Runge-Kutta methods.
The Euler method is a single-step method, this means that it only refer to one previous point and its derivative to determine the current value. Methods such as Runge-Kutta take some intermediate steps, for example, a half-step, to obtain a higher order method. Here we'll only take a look at the Euler method and a improved version of the Euler method, namely the Euler-Cromer method

4.2.1 The Euler method

[9] The Euler method was first published by Leonhard Euler around 1770 [12]. This method is explicit method, which means that it calculates the state of a system at a later time from the state of the system at the current time.
The idea behind his method was based on a particle that was moving in such a way that at time, t_0, its position was equal to x_0 and that, at this time, the velocity is a known quantity, v_0. By knowing t_0, t_1, x_0 and v_0 he could find the new position, x_0, at the time, t_0. The new position was given as

$$x_1 = x_0 + (t_1 - t_0)v_0 \tag{4.7}$$

If the difference between the next time, t_{i+1}, and the previous time, t_i, where $i = 0, 1, 2, \ldots$, is constant, then this is also known as Δt.
The Euler method that we use today based on a Taylor expansion

$$y(t + \Delta t) = y(t) + \Delta t \frac{dy}{dt} + \frac{1}{2}\Delta t^2 \frac{d^2y}{dt^2} + \frac{1}{3!}\Delta t^3 \frac{d^3y}{dt^3} + \ldots, \tag{4.8}$$

where we only use the first order

$$y(t + \Delta t) = y(t) + \Delta t \frac{dy}{dt} + \mathcal{O}(\Delta t^2), \tag{4.9}$$

This method gives us an error of $\mathcal{O}(\Delta t^2)$

The final form of the Euler method is

$$x(t_{i+1}) = x(t_i) + \Delta t f(t_i, x(t_i)) \tag{4.10}$$

If we are going to solve coupled differential equations like the one in Eq.4.6, we usually solve them like this

$$v(t_{i+1}) = v(t_i) + \Delta t f(t_i, x(t_i)) \tag{4.11}$$
$$x(t_{i+1}) = x(t_i) + \Delta t v(t_i) \tag{4.12}$$

4.2.2 The Euler-Cromer method

The Euler-Cromer is an improvement of the Euler method where we instead of using the velocity $v(t_i)$ in Eq. 4.12, we use the velocity for the new time step $v(t_{i+1})$ to find the position for the new time step $x(t_{i+1})$

$$v(t_{i+1}) = v(t_i) + \Delta t f(t_i, x(t_i)) \tag{4.13}$$

$$x(t_{i+1}) = x(t_i) + \Delta t v(t_{i+1}) \tag{4.14}$$

This is the method that has been used in all the numerical simulations in this thesis.

The Euler or Euler-Cromer methods is probably

4.3 Truncation error of the Euler method

The truncation error [15] is divided into two different errors.

1. Local truncation errors, which tells us the error caused by one iteration
2. global truncation errors, which tells us the cumulative error caused by many iterations.

4.3.1 Local truncation error

The local truncation error of the Euler method, is the error made in a single step. It is the difference between the numerical solution after one step, y_1, and the exact solution at time $t_1 = t_0 + h$. The numerical solution is given by

$$y_1 = y_0 + hf(t_0, y_0) \tag{4.15}$$

For the exact solution, we use the Taylor expansion mentioned above

$$y(t_0 + h) = y(t_0) + hy'(t_0) + \frac{1}{2}h^2 y''(t_0) + O(h^3). \tag{4.16}$$

The local truncation error (LTE) introduced by the Euler method is given by the difference between these equations

$$\text{LTE} = y(t_0 + h) - y_1 = \frac{1}{2}h^2 y''(t_0) + O(h^3). \tag{4.17}$$

This result is valid if y has a bounded third derivative
This shows that for small h, the local truncation error is approximately proportional to h^2. This makes the Euler method less accurate (for small h) than other higher-order techniques such as Runge-Kutta methods for which the local truncation error is proportial to a higher power of the step size.

4.3.2 Global truncation error

The global truncation error is the error at a fixed time t, after however many steps the methods needs to take to reach that time from the initial time. The global truncation error is the cumulative effect of the local truncation errors committed in each step. The number of steps is easily determined to be $(t - t_0)/h$, which is proportional to $1/h$, and the error committed in each step is proportional to h^2. Thus, it is to be expected that the global truncation error will be proportional to h

This intuitive reasoning can be made precise. The global truncation error is then

$$|\mathrm{GTE}| \leq \frac{hM}{2L}(e^{L(t-t_0)} - 1) \tag{4.18}$$

where M is an upper bound on the second derivative of y on the given interval and L is the Lipschitz constant of f. The precise form of this bound of little practical importance, as in most cases the bound vastly overestimates the actual error committed by the Euler method. What is important is that it shows that the global truncation error is (approximately) proportional to h. For this reason, the Euler method is said to be first order.

Part V

The 1D-Model

Chapter 5

Introduction

Spring-block models has been used to solve problems regarding friction and earthquakes since the 1960s. In the late 1960s R. Burridge and L. Knopoff [11, 16] used a numerical spring-block model to explore the role of friction along a fault as a factor in the earthquake mechanism. In [13] J. M. Carlson, J. S. Langer and B. E. Shaw use what they call the Burridge-Knopoff model to study the dynamics of earthquake faults. In [10] O. M. Braun, I. Barel and M. Urbakh propose a model for a description of dynamics of crack-like processes that occur at the interface between two blocks prior to the onset of frictional motion. Their model allows them to explain experimental observations in [23, 24] and predicts the effect of material properties on the dynamics of the transition to sliding. [17] This article describes the mechanism of precursor events; the mechanism was determined through an experiment and simulation by considering non-uniform normal loading

5.1 Side driven model

The 1D-model is a discretization of the the three dimensional PMMA system depicted i Figure 5.1. The upper block, which we call the slider, has a height, L_z, in the z-direction. A length, L_x, in the x-direction and a depth, L_y, in the y-direction. The length of the system is larger than the height, and much larger than the depth. The height is also larger than the depth. The slider is placed on top of another PMMA block, which we call the substrate.

The slider is divided into N equally sized blocks, that are connected to each other with springs. Each connecting spring has a spring stiffness that depends on the Young's modulus, length, hight and depth of the slider, and the number of blocks we divite the slider into. In this case the Young's modulus is that of PMMA. The Young's modulus is given as

$$E = \frac{\sigma}{\varepsilon} = \frac{F/A_0}{\Delta L/L_x} = \frac{FL_x}{\Delta LA_0} \tag{5.1}$$

where σ is the tensile stress and ε is the tensile strain. F is the force that is exerted on the object through the cross-section area A_0. The cross-section area of the slider is, $A_0 = L_yL_z$. ΔL is the blocks change in length. We now have to find some way to connect the spring

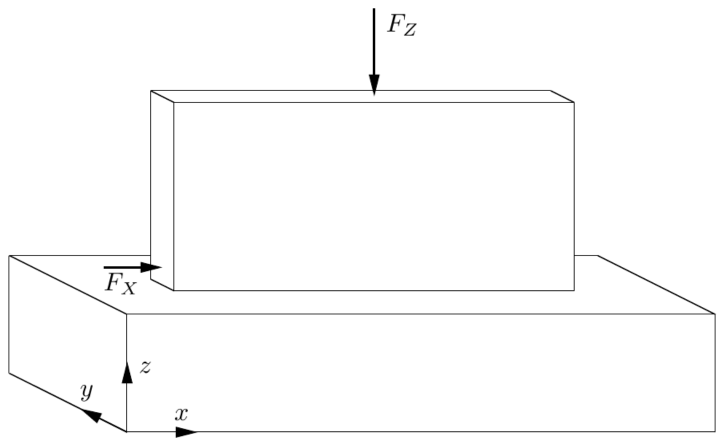

Figure 5.1: A three dimensional sketch of the PMMA system. The upper block is the slider, which moves, and the lower part is the substrate, which is at rest.

stiffness and the Young's modulus. Because the block is only deformed elastically we can use Hook's law

$$F = k'x \tag{5.2}$$

where F is the force, x is the distance from equilibrium, k' is the total spring stiffness. The spring stiffness for each spring connecting two blocks is given as $k = k'/(N-1)$. If we now combine the equation for the Young's modulus and Hook's law, we get an expression for the spring stiffness

$$k' = \frac{E\Delta L A_0}{x L_x} \tag{5.3}$$

If we now insert $A_0 = L_y L_z$, and $x = \Delta L$, we end up with the equation

$$k' = \frac{E L_y L_z}{L_x} \tag{5.4}$$

This is the total spring stiffness for the slider. The spring stiffness for each connecting spring is given as

$$k = \frac{E(N-1) L_y L_z}{L_x} \tag{5.5}$$

This spring stiffness form the basis for the interaction between the blocks.

The total mass for the slider is given as M, so the mass for each block is given as $m = M/N$.

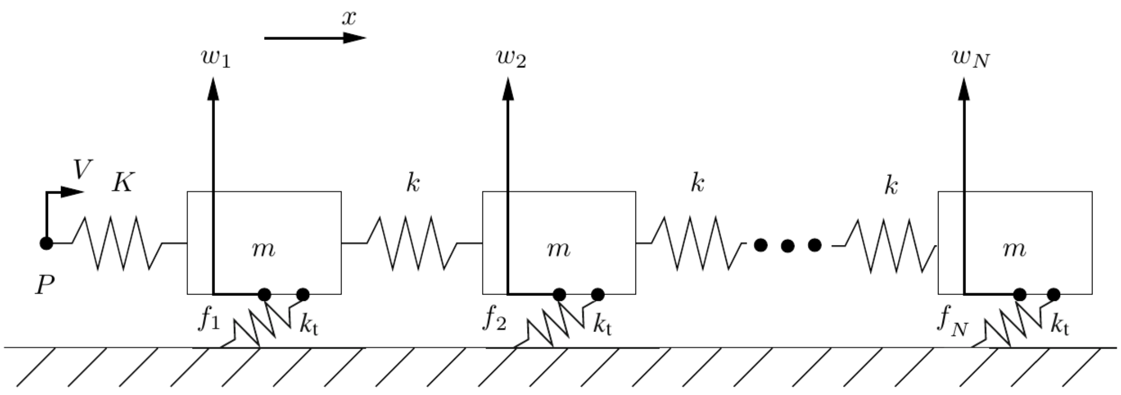

Figure 5.2: A side-driven one dimensional spring-block model. The point P is driven with a constant velocity, V, in the positive x-direction. A driving spring with spring stiffness, K, connects P and the first block. f_i and w_i, where $i = 1, 2, ..., N$, is respectively the friction force between the substrate and block i and the normal force acting on block i. m is the mass of each block.

5.2 Equations of motion

The equations of motion for a side driven system of N blocks is given by

$$m\ddot{u}_n = \begin{cases} k(u_2 - u_1) + F_X + F_1^\eta + f_1, & n = 1 \\ k(u_{n+1} - 2u_n + u_{n-1}) + F_n^\eta + f_n, & 2 \leq n \leq N-1 \\ k(u_{n-1} - u_N) + F_N^\eta + f_N, & n = N \end{cases} \tag{5.6}$$

The origin of these equations are Newton's second law of motion. The parts of the equations involving the position, u_n, is the force due to the interaction between neighboring blocks due to the connecting springs. F_η is the force due to the viscous damping and depends on the velocity of the blocks

$$F_n^\eta = \begin{cases} \eta(\dot{u}_2 - \dot{u}_1), & n = 1 \\ \eta(\dot{u}_{n+1} - 2\dot{u}_n + \dot{u}_{n-1}), & 2 \leq n \leq N-1 \\ \eta(\dot{u}_{n-1} - \dot{u}_N), & n = N \end{cases} \tag{5.7}$$

The force F_X is the driving force that pushes the whole system, in the side-driven case, it is connected to block number one. This driving force is produced by a spring with a spring stiffness, K. The spring is connected to the first block and a point, P, Figure 5.2. This point has a constant velocity V. The equation for the driving force is

$$F_X = K(Vt - u_1) \tag{5.8}$$

The normal force, p_n, on each block is given by the total normal force on the slider, F_N, divided by the total number of blocks

$$p_n = \frac{F_N}{N} \tag{5.9}$$

This gives a uniformly distributed normal force. We have also used non-uniformly distributed normal force given by the equation

$$p_n = \frac{F_N}{N}\left(1 + \frac{2n - N - 1}{N - 1}\theta\right), \tag{5.10}$$

where $\theta \in (-1, 1)$. For $\theta = 0$, we get the uniformly distributed normal force.
f_n is the friction force between block n and the substrate.

5.3 The friction model

If we look at the friction model from chapter 2

$$f_i = \begin{cases} f_s^{(i)}, & \text{when} \frac{dx}{dt} = 0 \\ -f_k^{(i)}, & \text{when} \frac{dx}{dt} > 0 \\ f_k^{(i)}, & \text{when} \frac{dx}{dt} < 0. \end{cases} \tag{5.11}$$

we can see that because of $f_s^{(i)}$ each block is at rest until it reaches the static friction threshold and suddenly starts moving. From experiments 3.3 **a)** we could see that the colors was not constant until slip. There was some change in the contact between the two blocks between slips. This indicates that the pure Amontons-Coulomb description is lacking some functionality.
One of the important parts of the numerical friction model is how we model the interaction between the base and the slider. The model we use here is the model depicted in Figure 5.3. In this thesis we'll use a friction law that uses the dynamic friction coefficient, μ_d, and the static friction coefficient, μ_s. μ_d will be used to model the friction force while a block is moving. μ_s will only be used to define the static friction threshold. The static friction threshold defines when the force pushing on a block is high enough to allow the block to move. μ_s is larger than μ_d.
At the beginning of the simulation, when $t_0 = 0$, all of the blocks will be at rest, and only static friction will be exerted on the system. The static friction force, f_n, follows a spring law, and the force is exerted thought a spring of stiffness k_t. This spring will have a length $u_n - u_n^{\text{stick}} = 0$ at t_0, where u_n is the position of block n, and u_n^{stick} is the position of the spring that is connected to the base. As driving force is exerted on the system, u_n will change, and the static friction force will follow the equation

$$f_n = -k_t(u_n - u_n^{\text{stick}}) \tag{5.12}$$

The static friction force will follow this equation until it reaches the static friction threshold, f_s, which follows the equation

$$f_s = \mu_s p_n \tag{5.13}$$

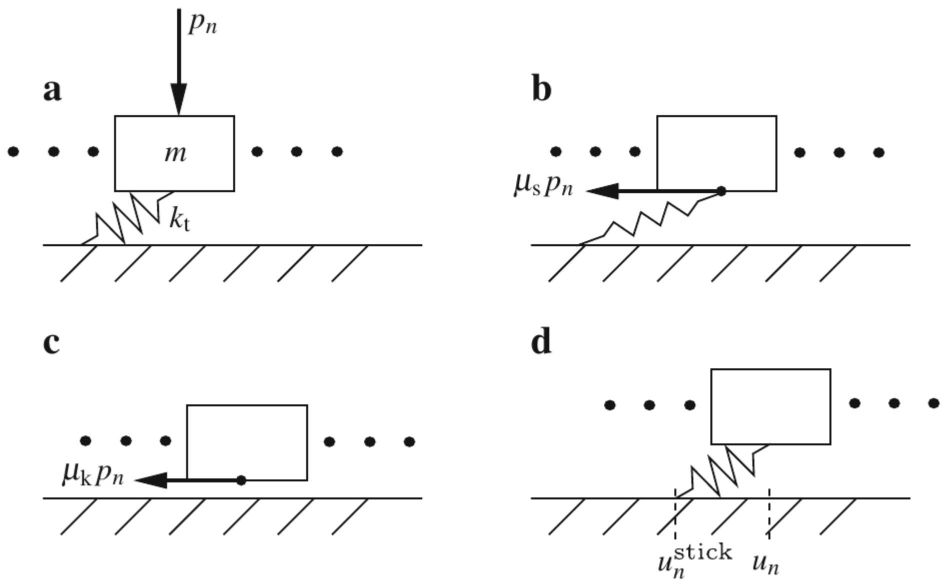

Figure 5.3: **(a)** The friction spring is attached to the base and to the block. The friction force acting on the block, is now given by the friction spring stiffness, k_t. **(b)** The friction spring force reaches the static friction threshold ,$\mu_s p_n$. **(c)** The static friction spring has broken, and the friction force on the block is now the dynamic friction force, $\mu_k p_n$. **(d)** The block comes to a halt, and the static friction spring attaches to the point u_n^{stick}.
The figure is from [3]

When $f_n \geqslant f_s$, we say that the spring breaks, and the block is subjected to a dynamic friction force

$$f_d = \mu_d p_n \tag{5.14}$$

This force has a sign in the opposite direction as the velocity. When the blocks velocity reaches zero, the static friction spring is reattached, and static friction is applied on the block. The position, u_n^{stick}, is chosen so that the total tangential force on the block is zero

$$u_n^{\text{stick}} = u_n - \frac{\tau_n}{k_t} \tag{5.15}$$

As the system is loaded tangentially, a finite region around the driving point is affected. As we can see from Figure 5.19, it is not just the first block that is affected.
How many blocks that is affected depends on the ratio between the block spring stiffness, k, and the friction spring stiffness, k_t. In Figure 5.20 we can see different rations the block spring stiffness and the friction spring stiffness. As we can see if $k_t > k$ more blocks are affected by the displacement of the first block, and if $k_t < k$ fewer blocks is affected by the displacement of the first block.

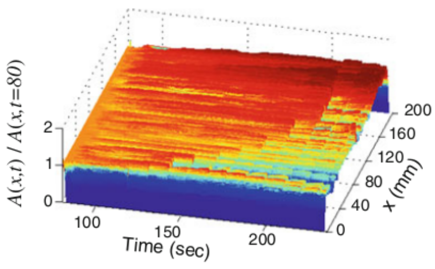

Figure 5.4: Her we can see the area of contact from an experiment. From [6]

5.4 Defining the precursor length

The precursor is a slip event, and the length of this event is called the precursor length, L_p. In experiments we can observe how long a slip event was by constantly measuring the contact area between the slider and the block. In Figure 5.4 we can measure the precursor event by looking at the sudden change in color along the x-axis for each time interval. In the numerical model we measure the precursor length by looking at how many nodes where affected by a slip event. If the node furthest from the beginning of the slip event is n_p and the first node is n_1, then the length of the precursor is

$$L_p = n_p - n_1 \tag{5.16}$$

If $n_p < n_N$, where n_N is the last node, then we say that we had a local slip event. If $n_p = n_N$, then the slip event has gone through all the nodes, and we have a global slip event.
Later in the results we'll use L_P/L as a measure of how far the slip event has come. Here L is the total length of the slider. If $L_P/L = 1$ this indicates that there has been a global slip.
An other way to discribe the precursor length is

$$L_p = (n_p/n_N)L \tag{5.17}$$

5.5 Asymmetric normal loading

The precursor length L_p is a measure on how many blocks, in the numerical model, that has reached its static friction threshold.

$$p_n = \frac{F_N}{N}\left(1 - \frac{2n - N - 1}{N - 1}\theta\right) \tag{5.18}$$

where p_n is the normal load on the n-th block and $\theta \in [-1, 1]$

$$F_T = \mu_k \sum_{n=1}^{n_p} p_n = \mu_k \frac{F_N}{N} \sum_{n=1}^{n_p} 1 - \frac{2n - N - 1}{N - 1}\theta \tag{5.19}$$

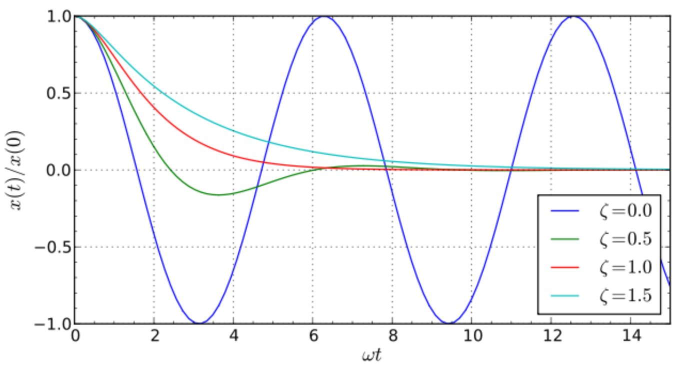

Figure 5.5: Here we can see a plot of different types of damping. The blue line is a system without damping, the green line is a under-damped system, the red line is a critically damped system and the cyan is an over-damped system.

$$F_T \approx \mu_k \frac{F_N}{N}\frac{N}{L}\int_0^{L_p}\left[1 - \frac{2(xN/L) - N - 1}{N-1}\theta\right]dx \tag{5.20}$$

if we now use that $N \pm 1 \approx N$ for large number of N, we get

$$F_T \approx \mu_k \frac{F_N}{L}\int_0^{L_p}\left[1 - \left(\frac{2x}{L} - 1\right)\theta\right]dx \tag{5.21}$$

$$F_T \approx \mu_k F_N \frac{L_p}{L}\left[1 + \theta\left(1 - \frac{L_p}{L}\right)\right] \tag{5.22}$$

5.6 Viscous damping

When we are working with spring systems we have to make sure that the system is behaving properly. By behaving properly, we mean that the system is neither under-damped nor over-damped. In Figure 5.5 we can see examples of different types of damping. For our system we would like to have a critically damped system.

If we ignore friction the equation of motion for our system is given as

$$m\ddot{u}_n = k(u_{n+1} - 2u_n + u_{n-1}) + \eta(\dot{u}_{n+1} - 2\dot{u}_n + \dot{u}_{n-1}) \tag{5.23}$$

We now use the wave equation

$$u_n(t) = e^{\zeta_\kappa t}e^{i\kappa n a}, \tag{5.24}$$

this is a method that is often used to find the stability for different numerical methods [21]. In the above equation, ζ_κ is a complex number and κ is a real spatial wave number. If we insert this into Eq.5.23, we get the equation

$$m\zeta_\kappa^2 = k(e^{i\kappa a} - 2 + e^{i\kappa a}) + \eta\zeta_\kappa(e^{i\kappa a} - 2 + e^{i\kappa a}) \tag{5.25}$$

We recognize that $e^{i\kappa a} - 2 + e^{i\kappa a}$ as version of the Euler's formula [9]

$$e^{i\kappa a} - 2 + e^{i\kappa a} = -4\sin^2\left(\frac{\kappa a}{2}\right) \tag{5.26}$$

By inserting this into Eq.5.25 we get

$$m\zeta_\kappa^2 + 4k\sin^2\left(\frac{\kappa a}{2}\right) + 4\eta\zeta_\kappa\sin^2\left(\frac{\kappa a}{2}\right) \tag{5.27}$$

If we solve this with respect to ζ_κ, we get the quadratic equation

$$\zeta_\kappa = \frac{-4\eta\sin^2\left(\frac{\kappa a}{2}\right) \pm \sqrt{16\eta^2\sin^4\left(\frac{\kappa a}{2}\right) - 16km\sin^2\left(\frac{\kappa a}{2}\right)}}{2m} \tag{5.28}$$

As mentioned earlier we are only interested in a system that is critically damped. The system is critically damped when Eq.5.27 has one solution for ζ_κ. ζ_κ has one solution when the square root in Eq.5.28 is zero

$$16\eta^2\sin^4\left(\frac{\kappa a}{2}\right) - 16km\sin^2\left(\frac{\kappa a}{2}\right) = 0 \tag{5.29}$$

$$\eta^2\sin^2\left(\frac{\kappa a}{2}\right) = km \quad \Rightarrow \quad \eta = \frac{\sqrt{km}}{|\sin\left(\frac{\kappa a}{2}\right)|} \tag{5.30}$$

The oscillations that are to be reduced have a wavelength $\lambda = 2a$ or a wave number $\kappa = 2\pi/\lambda = \pi/a$. Inserting this into Eq.5.30 lead to the critical damping coefficient

$$\eta_c = \sqrt{km} \tag{5.31}$$

Since the absolute value of sin is always smaller than one, choosing $\eta = \sqrt{km}$ will cause all other waves to be under-damped.

5.7 Tangential force

If we have a side-driven system with a shear force on block i given as $k(u_{n+1} - 2u_n + u_{n-1})$ which is the force from the two neighboring block $i+1$ and $i-1$. We also have the friction spring force $k_t(u_n - u_n^{\text{stick}})$. Under static friction these two forces are balancing each other out and we get this system of equations

$$k(u_{n+1} - 2u_n + u_{n-1}) - k_t(u_n - u_n^{\text{stick}}) = 0, \tag{5.32}$$

which is valid for all blocks except for the first and the last block. k is the stiffness of the spring between two blocks, the block spring stiffness, and k_t is the stiffness of the spring between a block and the substrate, the friction spring stiffness. We then introduce a new variable u'_n defined by $u_n = u'_n + u^0_n$, where u^0_n is the initial position of the n-th block. u'_n

is the displacement of each block. We also define τ_n^0, the initial shear force, which is given by the initial positions of the blocks

$$\tau_n^0 = k(u_{n+1}^0 - 2u_n^0 + u_{n-1}^0) \tag{5.33}$$

By inserting the expression for the displacement and the initial shear force into eq.(5.32) we end up with the equation

$$k(u'_{n+1} - 2u'_n + u'_{n-1}) - k_t u'_n + \tau_n^0 - k_t(u_n^0 - u_n^{\text{stick}}) = 0 \tag{5.34}$$

Because the initial system is at rest, the shear force for neighboring blocks,$i+1$ and $i-1$, and the friction force from the substrate on block i has to cancel out each other. $\tau_n^0 = k_t(u_n^0 - u_n^{\text{stick}})$. We then end up with system of equations described by the displacement of each block

$$k(u'_{n+1} - 2u'_n + u'_{n-1}) - k_t u'_n = 0 \tag{5.35}$$

By introducing a constant $a = L/(N-1)$, where $N \gg 1$, and multiplying and dividing the block-spring expression with it, we end up with en expression that looks very much like a discretized differential equation.

$$ka^2 \frac{u'_{n+1} - 2u'_n + u'_{n-1}}{a^2} - k_t u'_n = 0 \tag{5.36}$$

With differentials the the equation becomes

$$ka^2 \frac{\partial u'(x)}{\partial x^2} - k_t u'(x) = 0 \tag{5.37}$$

this differential equation has the general solution

$$u'(x) = Ae^{x/l_0} + Be^{-x/l_0} \tag{5.38}$$

where l_0, which is the characteristic length, is given as

$$l_0 = \sqrt{\frac{k}{k_t}} a \tag{5.39}$$

When the slider is moving, the tangential force on each block is given by

$$\begin{aligned} \tau_n &= k(u_{n+1} - 2u_n + u_{n-1}) \\ &= k(u'_{n+1} - 2u'_n + u'_{n-1}) + \tau_n^0 \end{aligned} \tag{5.40}$$

By using the same trick as we did in eq.(5.36), we end up with a equation for the tangential force for the system

$$\tau(x) = ka^2 \frac{\partial u'(x)}{\partial x^2} + \tau^0(x) \tag{5.41}$$

which has the same analytical solution as eq.(5.37)

$$\tau(x) = \frac{ka^2 l_0^2}{L^2}\left(Ae^{x/l_0} + Be^{-x/l_0}\right) + \tau^0(x) \tag{5.42}$$

We now have to find expressions for A and B by using our boundary conditions. We know that at $x = 0$, this is the first block, $\tau(0) = \mu_k p_1$, and that at $x = L$, this is the last block, $\tau(L) = \tau(L)$. We also use that $l_0/L \ll 1$, we then get

$$\begin{aligned}\tau(L) &= \frac{ka^2 l_0^2}{L^2}\left(Ae^{L/l_0} + Be^{-L/l_0}\right) + \tau^0(L) \\ &\approx \frac{ka^2 l_0^2}{L^2}(Ae^{L/l_0}) + \tau^0(L) \\ &= \tau^0(L),\end{aligned} \tag{5.43}$$

we see that for this to be true, $A = 0$. For $x = 0$ we get

$$\begin{aligned}\tau(0) &= \frac{ka^2 l_0^2}{L^2}B + \tau^0(0) \\ &= \mu_s p_1\end{aligned} \tag{5.44}$$

We then end up with the final expression for the tangential force

$$\tau(x) = \left(\mu_s p_1 - \tau^0(x)\right)e^{-x/l_0} + \tau^0(x) \tag{5.45}$$

5.8 Initial Shear Force

In some cases we would like to do a simulation where the system is initialized with some kind of initial shear force profile. If the slider is squeezed in the z-direction due to the pressure from the normal force the slider will get an initial shear force profile due to the expansion in the x and y-direction. In the case of the one-dimensional system, the system will only get the expansion in the x-direction. In Figure **??**
The total tangential load, F_T, on our system is

$$F_T = \sum_{n=1}^{N} \tau_n \tag{5.46}$$

If we go from the limit $N \to \infty$, we can rewrite the sum to an integral

$$\sum_{n=1}^{N} \to \frac{N}{L}\int_0^L \tau(x)dx, \quad n \to xN/L \tag{5.47}$$

We spilt the integral into two parts. One part up to the length up to the precursor length L_p, and one for the length after the precursor length. In Figure 5.6 we can see that before the L_p the normalized shear force for each block is fluctuating around μ_k. So the total shear force up to L_p is $\int_0^{L_p} \mu_k p(x)dx$. The tangential load is then

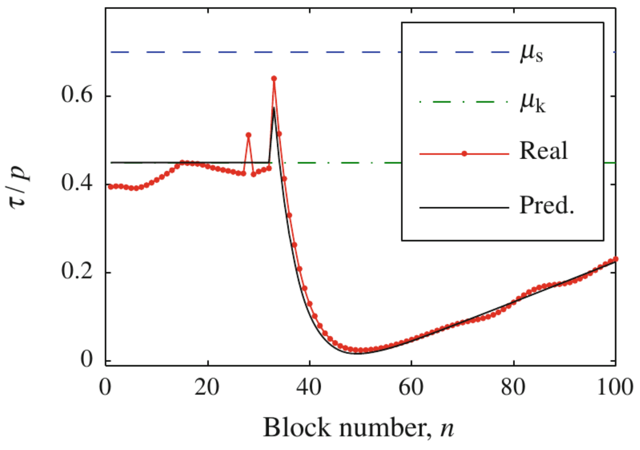

Figure 5.6: Here we can see the shear force τ normalized by the normal force. The precursor length, L_p, is the length between block 1 and block 33

$$\begin{aligned} F_T &= \frac{N}{L}\left[\int_0^{L_p} \tau(x)dx + \int_{L_p}^{L} \tau(x)dx\right] \\ &= \frac{N}{L}\left[\int_0^{L_p} \mu_k p(x)dx + \int_{L_p}^{L} \alpha p(L_p) - \tau^0(x)e^{-\frac{x-L_p}{l_0}} + \tau^0(x)dx\right], \end{aligned} \tag{5.48}$$

α is the average between the static friction coefficient and the dynamic coefficient, $\alpha = (\mu_s + \mu_k)/2$. The normal load, p, on each block is constant and uniformly distributed

$$p(x) = p = F_N/N = constant \tag{5.49}$$

We use a simple distribution for the initial shear force profile given by

$$\tau^0(x) = \beta p \frac{2(x - L/2)}{L}, \tag{5.50}$$

in Figure 5.7 we can see the plot for $\beta = 0.45$, $\beta = 0.225$ and $\beta = 0$. When $\beta = 0$ there is no initial shear force. By inserting eq.(5.49) and eq.(5.50) into eq.(5.48) we get the expression

$$\begin{aligned} F_T = \frac{N}{L}\Bigg[&\int_0^{L_p} \mu_k p dx + \int_{L_p}^{L} \left(\alpha p - \beta p \frac{2(x-L/2)}{L}\right) e^{-\frac{x-L_p}{l_0}} dx \\ &+ \int_{L_p}^{L} \beta p \frac{2(x-L/2)}{L} dx\Bigg] \end{aligned} \tag{5.51}$$

By solving this equation we end up whit the final expression for a side driven system with an initial shear force profile given by eq. (5.50)

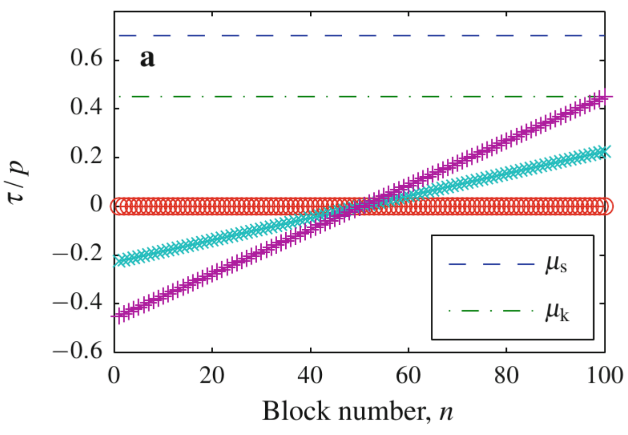

Figure 5.7: Here we have three different initial shear force profiles. The steepest one is for $\beta = 0.45$, the flat one is for $\beta = 0$ and the one between is for $\beta = 0.225$

$$F_T(L_p) = F_N \left[\mu_k \frac{L_p}{L} + 2\beta \frac{l_0^2}{L^2} \left(e^{-\frac{L-L_p}{l_0}} - 1 \right) + \beta \frac{(L-L_p)L_p}{L^2} \right. \\ \left. + \frac{l_0}{L} \left(\beta \left(1 + e^{-\frac{L-L_p}{l_0}} - 2\frac{L_p}{L} \right) + \alpha \left(1 - e^{-\frac{L-L_p}{l_0}} \right) \right) \right] \tag{5.52}$$

If we now insert $L_p = L$, which means that the precursor has reached the rightmost block, we end up with $F_T(L) = \mu_k F_N$.

5.9 Top driven

The top driven model shares is quite similar to the side driven model. The only difference is how the system is driven. In Figure 5.8 we can see that all the blocks are connected to larger flat block. This larger flat block is then moved at a constant velocity V. To get the same driving force, with the same driving velocity, as we get in the side driven system, the driving spring constant K has to be divided by the number of blocks. The driving spring constant for each block then becomes

$$K_n = \frac{K}{N}, \qquad n = 1, 2, \ldots, N \tag{5.53}$$

With the new driving spring constant, the new equation of motion then becomes

$$m\ddot{u}_n = \begin{cases} k(u_2 - u_1) + F_1^X + F_1^\eta + f_1, & n = 1 \\ k(u_{n+1} - 2u_n + u_{n-1}) + F_n^X + F_n^\eta + f_n, & 2 \le n \le N-1 \\ k(u_{n-1} - u_N) + F_N^X + F_N^\eta + f_N, & n = N \end{cases} \tag{5.54}$$

where F_n^X is the driving force on each block, which is given as

$$F_n^X = K_n(Vt - u_n), \qquad n = 1, 2, \ldots, N \tag{5.55}$$

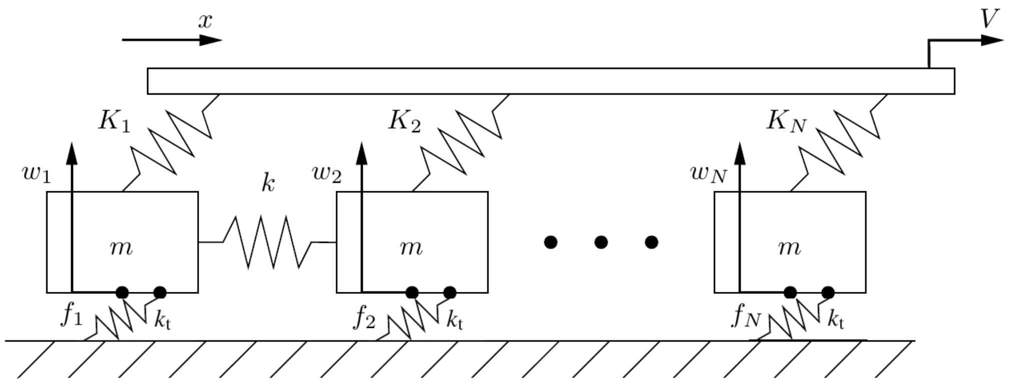

Figure 5.8: A top-driven one dimensional spring-block model. All the blocks are connected to a "driving" block, which is driven with a constant velocity $V.K_n$, f_n and w_n, where $n = 1, 2, ..., N$, is respectively the driving spring constant, the friction force between the substrate and block i and the normal force acting on block i. m is the mass of each block.

The code for the top-driven one-dimensional model is attached in Appendix B.5

5.10 Validation

5.10.1 The side driven model

One of the difficult parts when it comes to numerical models is to know if the model is correct. Different ways to validate the model might be to compare the results with experiment, or with results from similar simulations. If one is really lucky, some of the problems might have analytical solutions.
In our case some of the models have an analytical solution, and we'll start with those.

The asymmetric normal loading

The asymmetric normal loading was introduced in chapter two, and an analytical solution was given earlier in this chapter. If the implementation of the numerical model is correct, the numerical solution should be close to the analytical solution when we change the way we load the model. In this case we have used the same values as Maegawa et al. [17]. In Figure 5.9 we can see how good the correspondence is between the analytical model and the numerical model. We have used the values $\theta = 0.8333$, $\theta = 0$ and $\theta = -0.8333$.
If we now compare the numerical results with the experimental results from Maegawa et al. Figure. 5.10 we can see that the correspondence between the numerical solution and the experimental results in quit bad. Maegawa et al. also conducted some numerical simulations Figure. 5.11, and as we can see their numerical results was also quite different from the experimental results. As they write in [17]: The reason for the difference is not clear; it is possible that the mechanism underlying the propagation of precursors changes at around $L_p/L = 0.7$, or the partial normal load around the leading edge may be smaller than the expected one due to the unexpected curvature of the contact surfaces.

The initial share force
As with the asymmetric normal loading we also have an analytical solution for the Initial share force. By using the initial share force we can initiate our system with a desired share force. In the case of the analytical model this is done by changing β, the higher β is the greater becomes the initial share force. In Figure 5.12 both the analytical and the numerical results has been plotted. We can see that there is a good correspondence between the analytical solution and the numerical model.
Time evolution of the tangential load F_X
In Figure 5.13 and 5.14 we can see how the tangential load evolves as a function of time. As we can see the results from the model in this thesis corresponds quite well with the results from the simulations of Maegawa et al.

5.11 Results

5.11.1 Variation of the static and dynamic friction coefficient in the side-driven model

The results for the variation of μ_s and μ_k is plottet in Figure. 5.15, 5.16, 5.17 and 5.18. Here we have used the values

1. $\mu_s = 1.4$ and $\mu_k = 0.9$
2. $\mu_s = 0.35$ and $\mu_k = 0.225$
3. $\mu_s = 0.85$ and $\mu_k = 0.6$
4. $\mu_s = 0.55$ and $\mu_k = 0.3$

In the first case we have doubled the values of μ_s and μ_k compared to what we normally have used. In the second case we have halved the values of μ_s and μ_k compared to what we normally have used. In the third case we have added 0.15 to each of the values of μ_s and μ_k compared to what we normally have used. And in the last case we have subtracted 0.15 to each of the values of μ_s and μ_k compared to what we normally have used.
First let us analyze the first case. Her we have that $\mu_s/\mu_k = 1.4/0.9 \approx 1.56$ and $F_X(t)/P$ reach the value of μ_k after about 5.2s slope between the curve and the time axis is $0.9/5.2 \approx 0.17$.
For the second case we get these results. Her we have that $\mu_s/\mu_k = 0.35/0.225 \approx 1.56$ and $F_X(t)/P$ reach the value of μ_k after about 1.3s slope between the curve and the time axis is $0.225/1.3 \approx 0.17$.
If we compare the first case and the second case, we see that we got the same answers. This indicates that as long as we keep the relationship between μ_s and μ_k constant, well end up with the same slope.
Now lets take a look at the third case. Her we have that $\mu_s/\mu_k = 0.85/0.6 \approx 1.42$ and $F_X(t)/P$ reach the value of μ_k after about 3.55s slope between the curve and the time axis is $0.6/3.55 \approx 0.17$.
Now lets take a look at the third case. Her we have that $\mu_s/\mu_k = 0.55/0.3 \approx 1.833$ and $F_X(t)/P$ reach the value of μ_k after about 1.75s slope between the curve and the time axis is $0.3/1.75 \approx 0.17$.
An quite interesting observation of the above analysis of the data suggests that the relation between μ_k and the time it takes to reach this value follows a linear law.

$$\mu_k \approx 0.17t \qquad \Rightarrow t = \frac{\mu_k}{0.17} \tag{5.56}$$

where t is the time it takes to reach μ_k.

5.11.2 Different ratios between k_t and k

Here we'll look at the results from Figure 5.20. Here we have plotted four different relations between the friction spring stiffness and the spring stiffness of the coil spring between the blocks. We have used the ratios

1. $k_t/k = 0.1$
2. $k_t/k = 0.5$
3. $k_t/k = 2$
4. $k_t/k = 10$

A relation of $k_t/k < 1$ means that the friction spring is stiffer than the spring between the blocks. A relation of $k_t/k > 1$ means that the spring between the blocks is stiffer than the friction spring.
We observe that when $k_t/k = 0.1$ the number of affected blocks is approximately 17. For $k_t/k = 0.5$ the number of affected blocks is approximately 8. For $k_t/k = 2$ the number of affected blocks is approximately 5. For $k_t/k = 10$ the number of affected blocks is approximately 3.
From the above numbers, we can see that when $k_t/k \to 0$ the number of affected nodes $\to N_x$. And when $k_t/k \to \infty$ the number of affected nodes $\to 0$.

5.11.3 Variation of μ_k and μ_s and the affect of L_P/L-F_T/F

In Figure 5.21, 5.22, 5.23 and 5.24, we have plotted the relationship between the precursor length and the shear force. In each plot there is a circle and a line. The circle represents the numerical results, and the black line represents the analytical solution.
we have also here used the values

1. $\mu_s = 1.4$ and $\mu_k = 0.9$
2. $\mu_s = 0.35$ and $\mu_k = 0.225$
3. $\mu_s = 0.85$ and $\mu_k = 0.6$
4. $\mu_s = 0.55$ and $\mu_k = 0.3$

In the first case, where $\mu_s = 1.4$ and $\mu_k = 0.9$, we see that when $L_P/L = 1$, $F_T/F_N = \mu_k = 0.9$ In the second case, where $\mu_s = 0.35$ and $\mu_k = 0.225$, we see that when $L_P/L = 1$, $F_T/F_N = \mu_k = 0.225$ In the first case, where $\mu_s = 0.85$ and $\mu_k = 0.6$, we see that when $L_P/L = 1$, $F_T/F_N = \mu_k = 0.6$ In the first case, where $\mu_s = 0.55$ and $\mu_k = 0.3$, we see that when $L_P/L = 1$, $F_T/F_N = \mu_k = 0.3$
We can see that the linear relationship between L_P/L and F_T/F_N is

$$F_T/F_N = \mu_k L_P/L \tag{5.57}$$

This indicates that relationship between L_P/L and F_T/F_N is only dependent on μ_k
The code for the one-dimensional side-driven model is attached in Appendix B.4

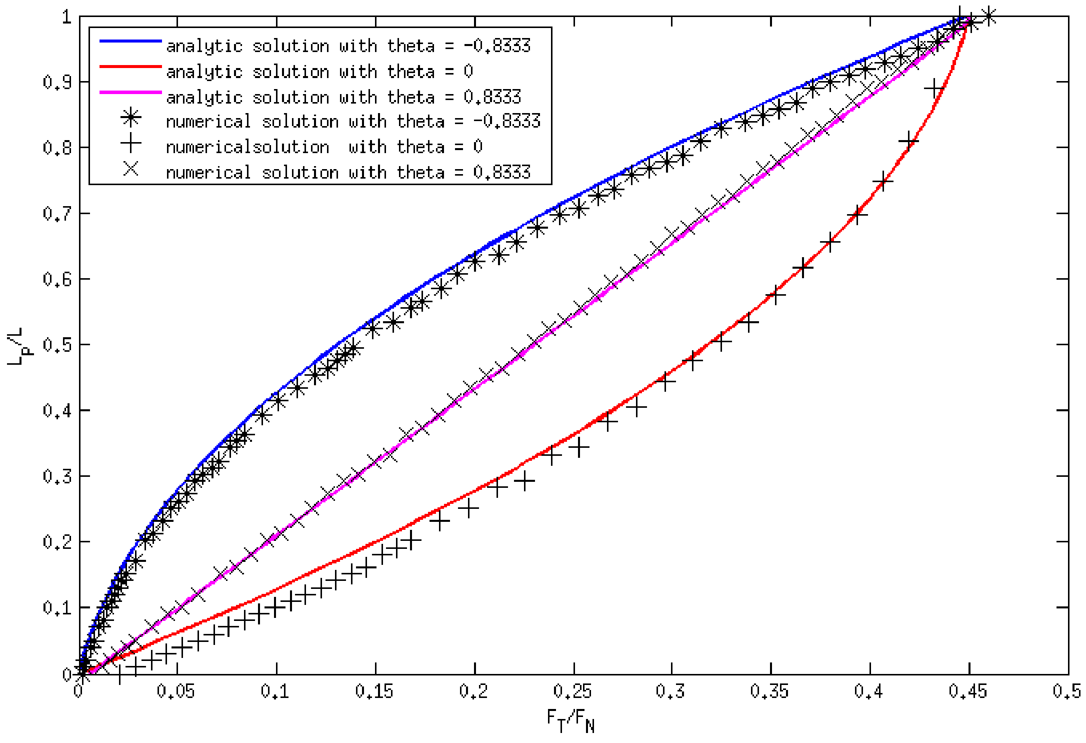

Figure 5.9: Here we can see a side driven system with asymmetric normal loading. We have used $\theta = 0.8333$, $\theta = 0$ and $\theta = -0.8333$ for both the numerical and analytical curves.

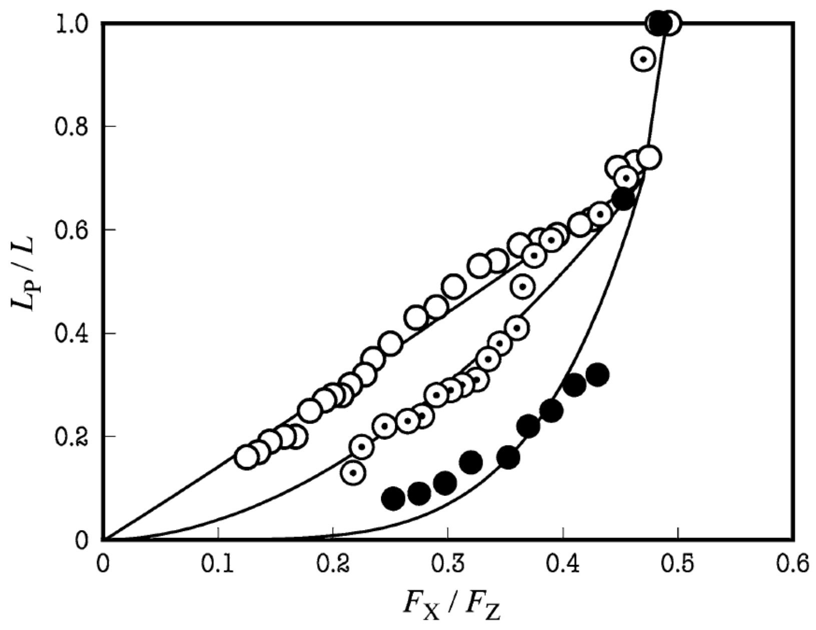

Figure 5.10: Experimental $L_p/L - F_T/F_N$ results from Maegawa et al. [17]

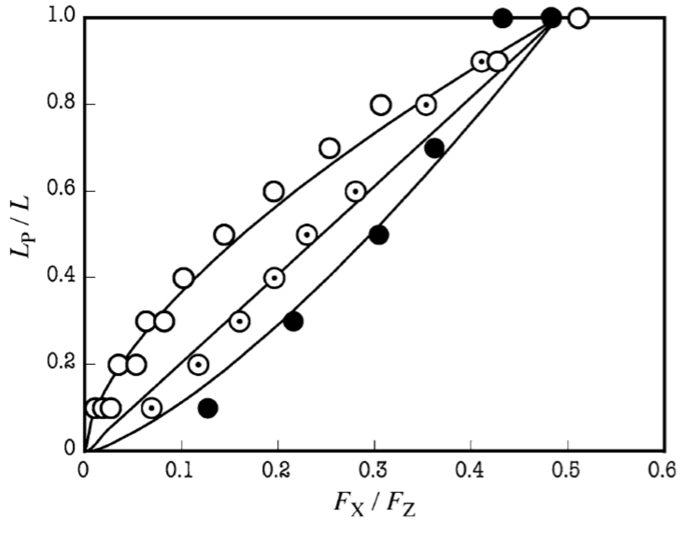

Figure 5.11: Numerical $L_p/L - F_T/F_N$ results from Maegawa et al. [17]

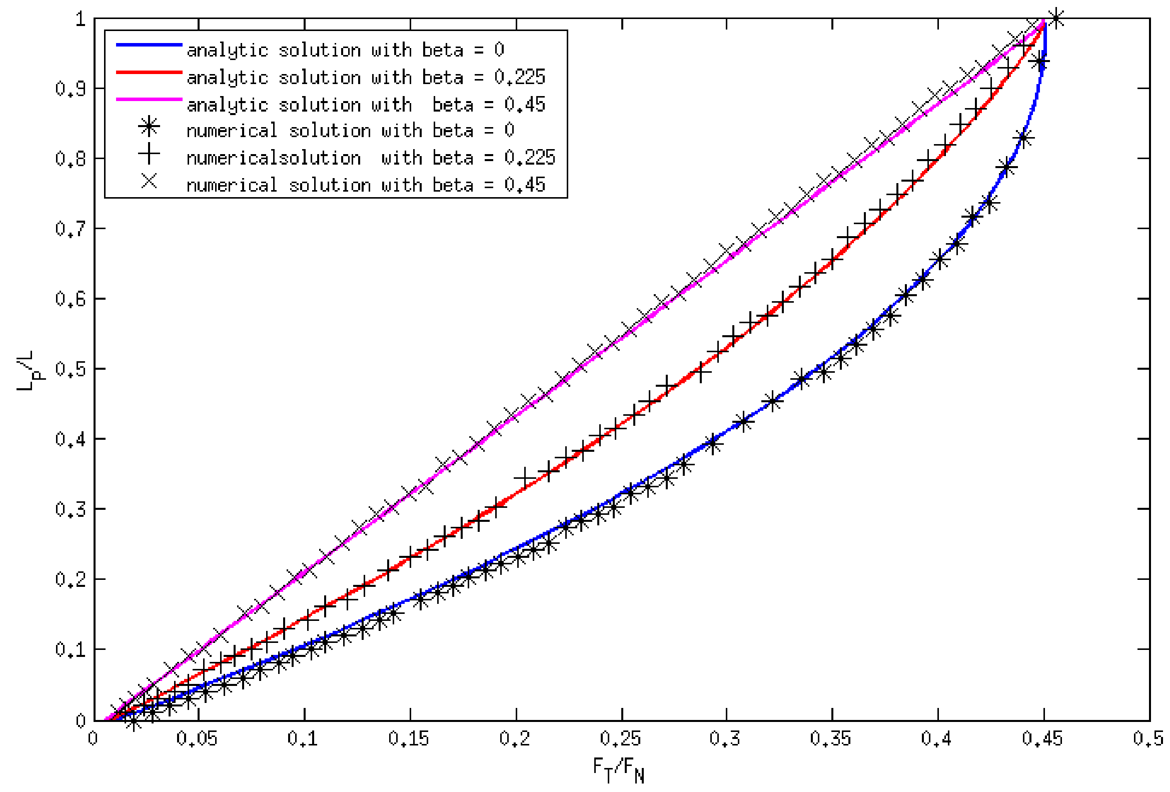

Figure 5.12: $L_p/L - F_T/F_N$ plot for a system with an initial shear force profile. $\beta = 0$, $\beta = 0.225$ and $\beta = 0.45$

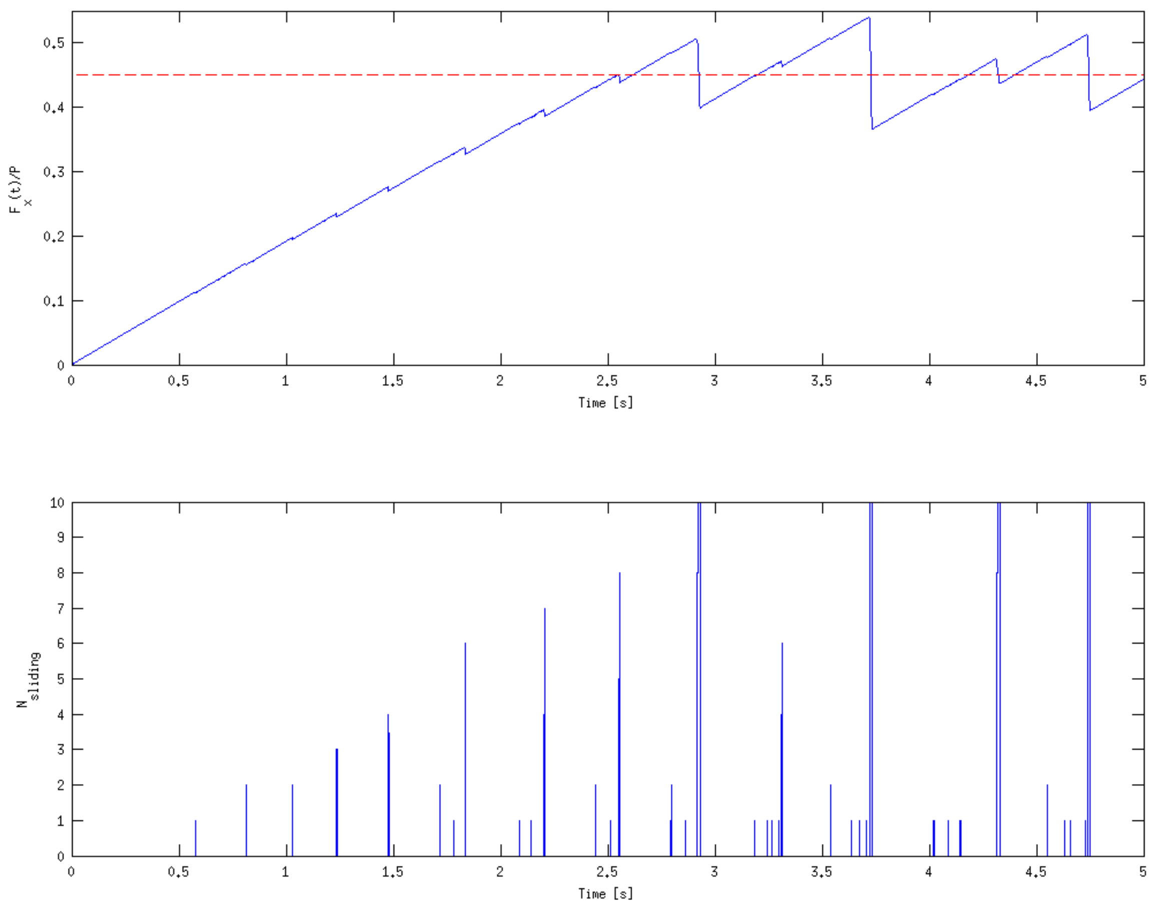

Figure 5.13: F_x plot for $N = 10$ from a numerical simulation

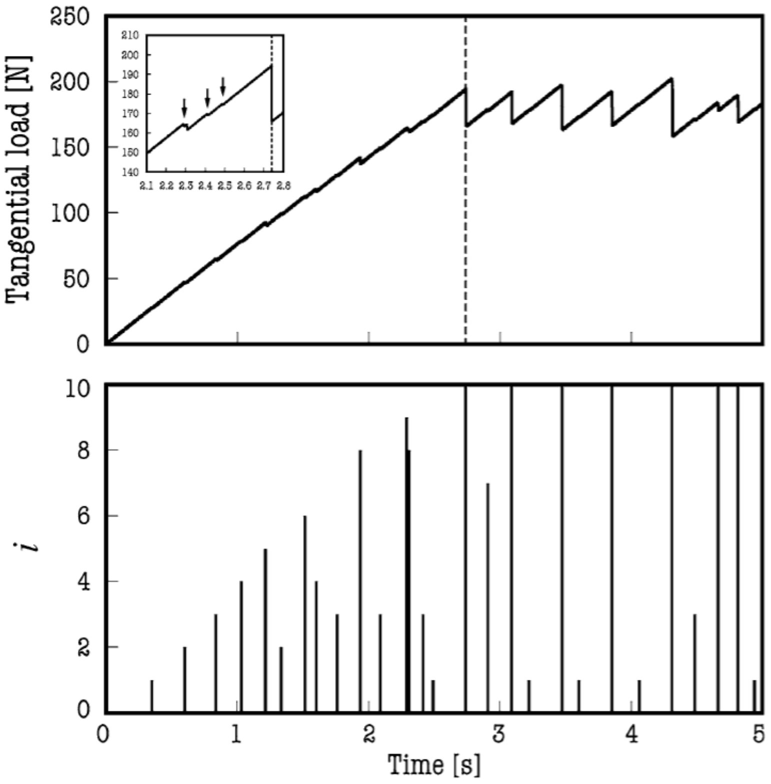

Figure 5.14: F_x plot for $N = 10$ from a numerical simulation Maegawa et al.

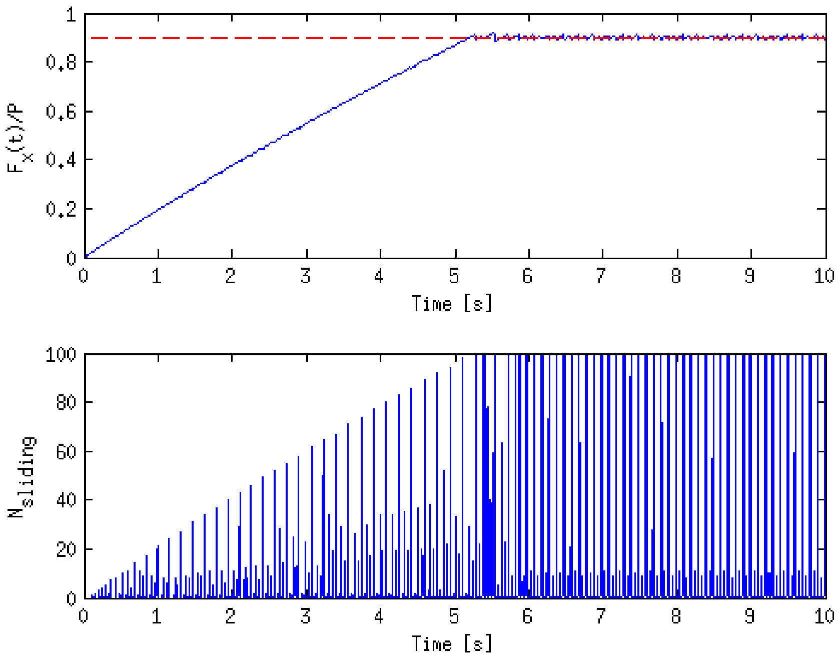

Figure 5.15: Results from a side-driven simulation. Here $\mu_s = 1.4$ and $\mu_k = 0.9$. The simulation time was $T = 10$s and the number of blocks was $N = 100$.

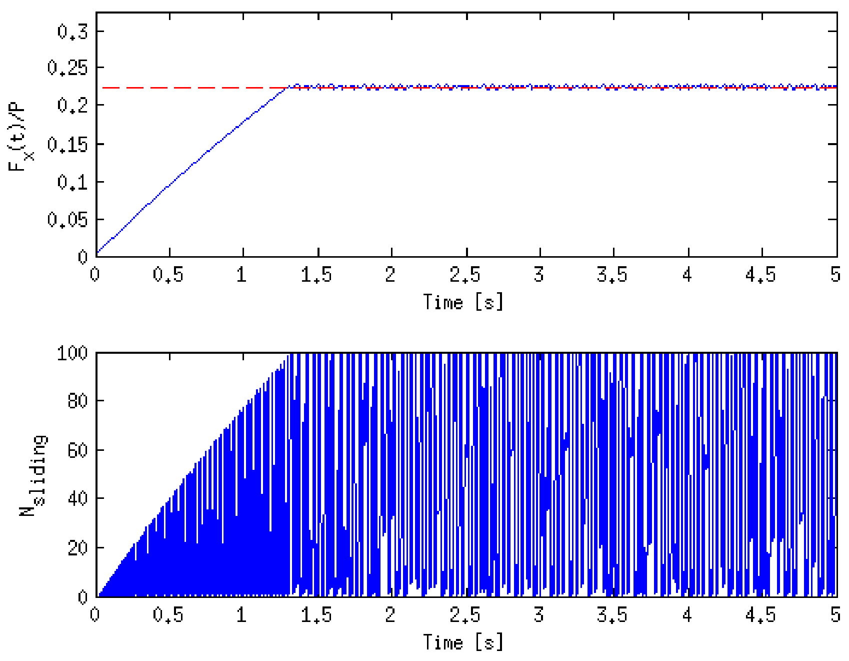

Figure 5.16: Results from a side-driven simulation. Here $\mu_s = 0.35$ and $\mu_k = 0.225$. The simulation time was $T = 5$s and the number of blocks was $N = 100$.

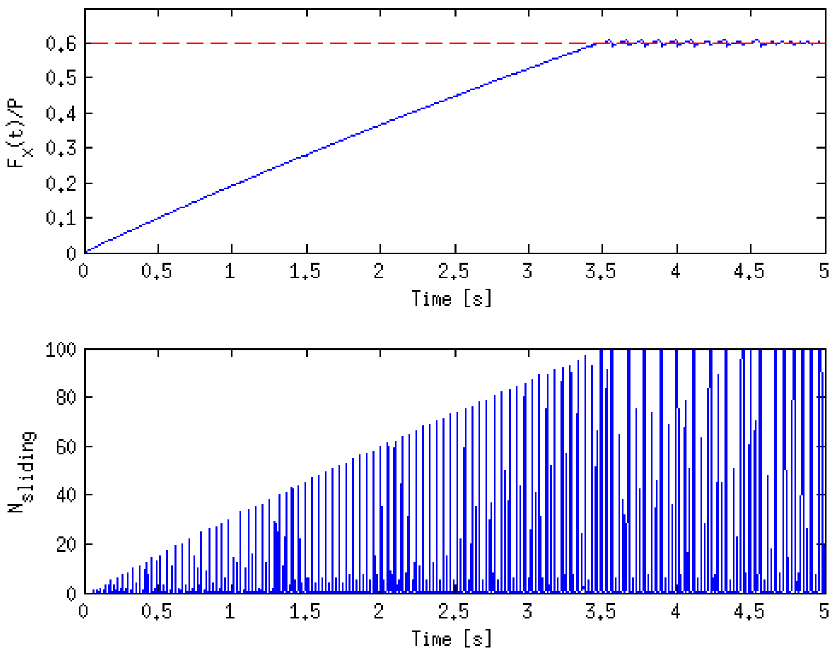

Figure 5.17: Results from a side-driven simulation. Here $\mu_s = 0.85$ and $\mu_k = 0.6$. The simulation time was $T = 5$s and the number of blocks was $N = 100$.

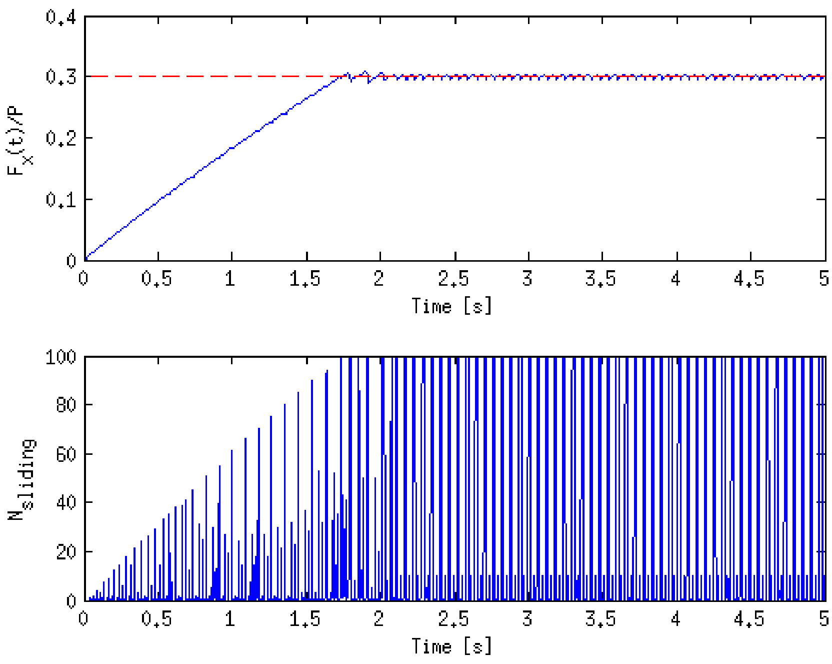

Figure 5.18: Results from a side-driven simulation. Here $\mu_s = 0.55$ and $\mu_k = 0.3$. The simulation time was $T = 5$s and the number of blocks was $N = 100$.

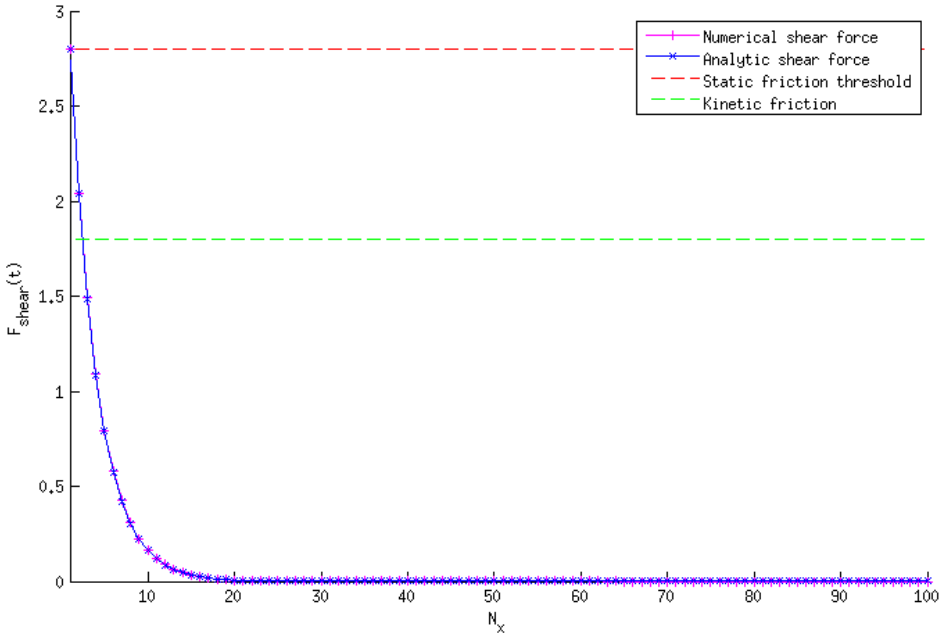

Figure 5.19: Here we can see the shear force F_{shear} plotted against the number of blocks, N_x, at initiation of the first precursor. The pink line is the numerical solution of the system, and the blue line is the analytical solution of the system as shown in eq.5.45 In this case $N_x = 100$.

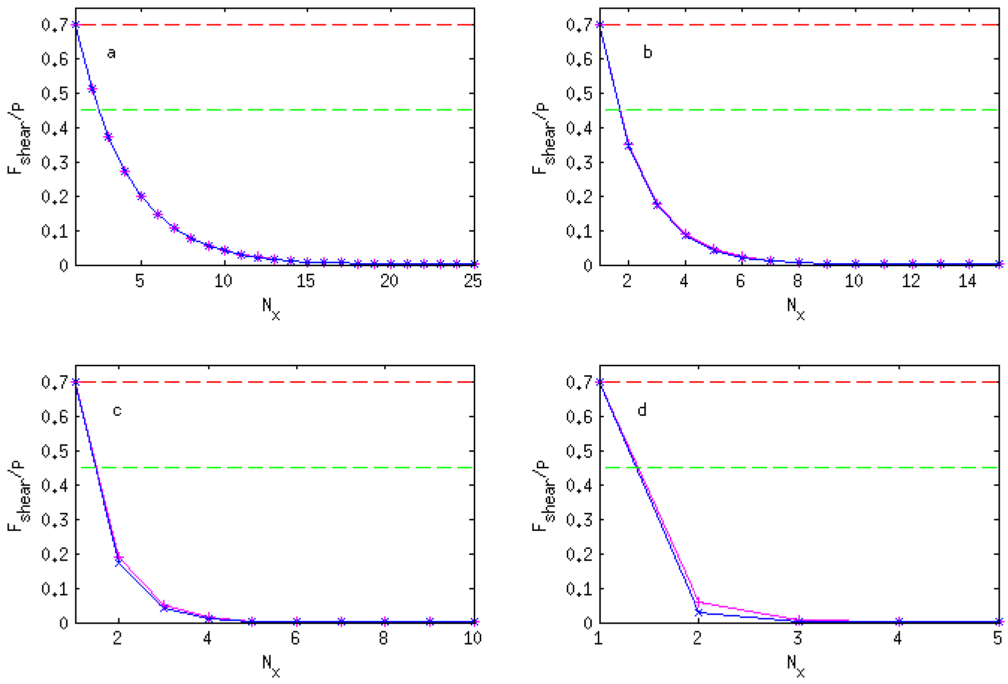

Figure 5.20: This figure shows the same as Figure 5.19, but here we have displayed more ratios. **(a)** $k_t/k = 0.1$. **(b)** $k_t/k = 0.5$. **(c)** $k_t/k = 2$. **(d)** $k_t/k = 10$. $N_x = 100$ in each case

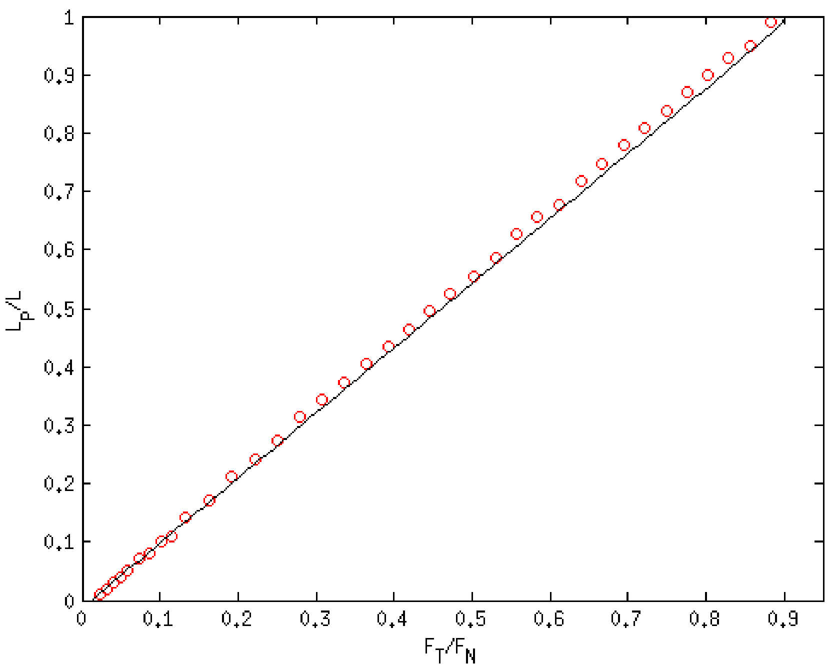

Figure 5.21: Side-driven L_p/L and F_T/F_N ($\mu_s = 1.4$ and $\mu_k = 0.9$). The red circles are the numerical results, and the black line is the analytical solution.

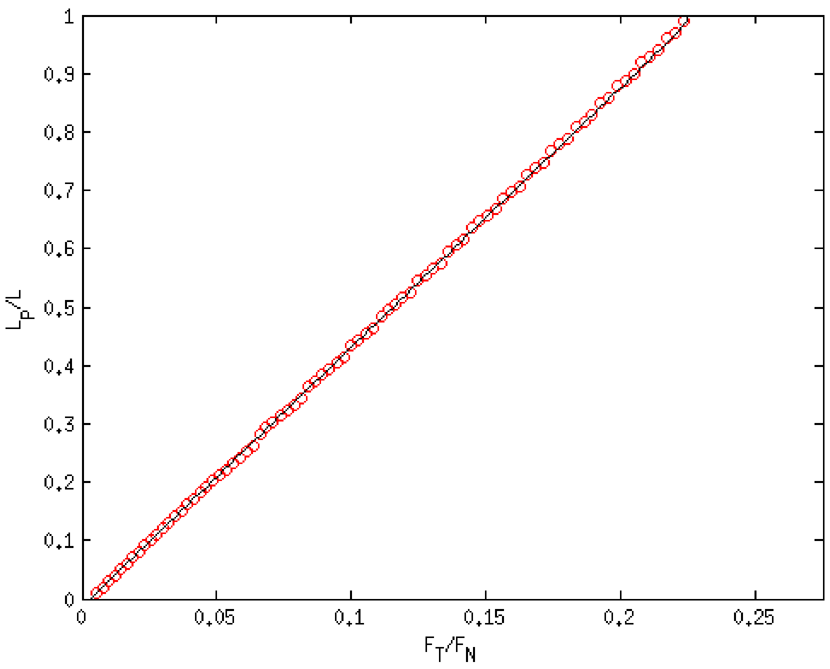

Figure 5.22: Side-driven L_p/L and F_T/F_N ($\mu_s = 0.35$ and $\mu_k = 0.225$). The red circles are the numerical results, and the black line is the analytical solution.

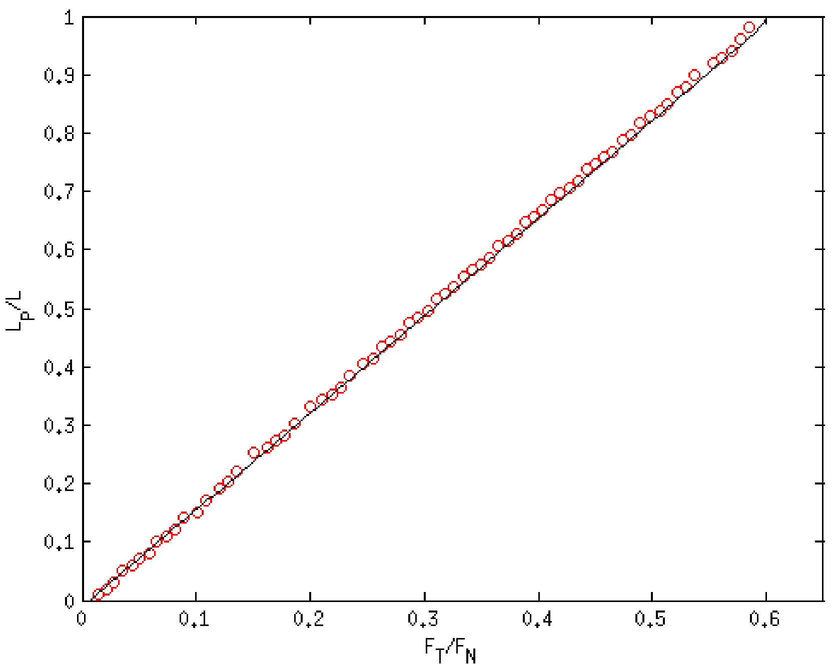

Figure 5.23: Side-driven L_p/L and F_T/F_N ($\mu_s = 0.85$ and $\mu_k = 0.6$). The red circles are the numerical results, and the black line is the analytical solution.

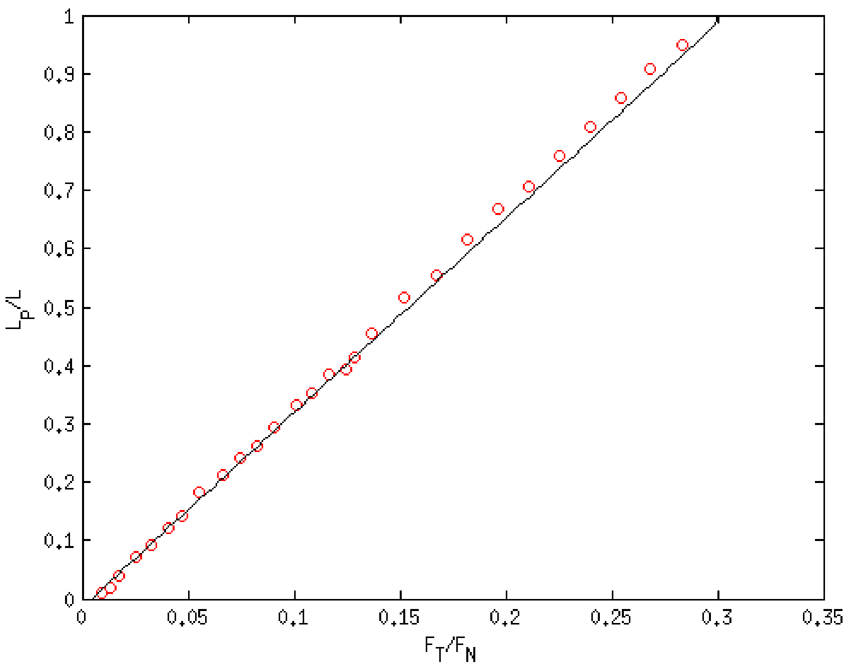

Figure 5.24: Side-driven L_p/L and F_T/F_N ($\mu_s = 0.55$ and $\mu_k = 0.3$). The red circles are the numerical results, and the black line is the analytical solution.

Part VI

The 1D-Unloading-Model

Chapter 6

The 1D-Unloading-Model

6.1 The model

In this part of the thesis we'll look at a 1D spring-block model where the normal force is a function of time. The block dynamics of this system is the same as for the side-driven model. In Figure 6.1 we can see how the system is set up. The normal force starts out at

$$F_X(t) = \begin{cases} K(Vt - u_1), & t < T_U^{\text{start}} \\ K(VT_U^{\text{start}} - u_1), & t > T_U^{\text{start}} \end{cases} \tag{6.1}$$

where T_U^{start} is the time we start to reduce the value of the normal force. The normal force is given as

$$F_N(t) = \begin{cases} F_N^{\text{start}}, & t < T_U^{\text{start}} \\ F_N^{\text{start}} - \nu t, & T_U^{\text{start}} \leq t < T_U^{\text{end}} \\ F_N^{\text{end}}, & t > T_U^{\text{end}} \end{cases} \tag{6.2}$$

Where F_N^{start} is the initial normal force, F_N^{end} is the normal force we would like to end up with at the end of the simulation, T_U^{end} is the time when we stop reducing the normal force ν is the reduction slope of the normal force and is given as

$$\nu = \frac{F_N^{\text{start}} - F_N^{\text{end}}}{T_U^{\text{end}} T_U^{\text{start}}} \tag{6.3}$$

The dynamic friction force is now time dependent and is given as

$$f_d = \mu_d p_n(t), \tag{6.4}$$

where $p_n(t) = F_N(t)/N$. The static friction threshold is given as

$$f_s = \mu_s p_n(t) \tag{6.5}$$

The equation of motion for this system is

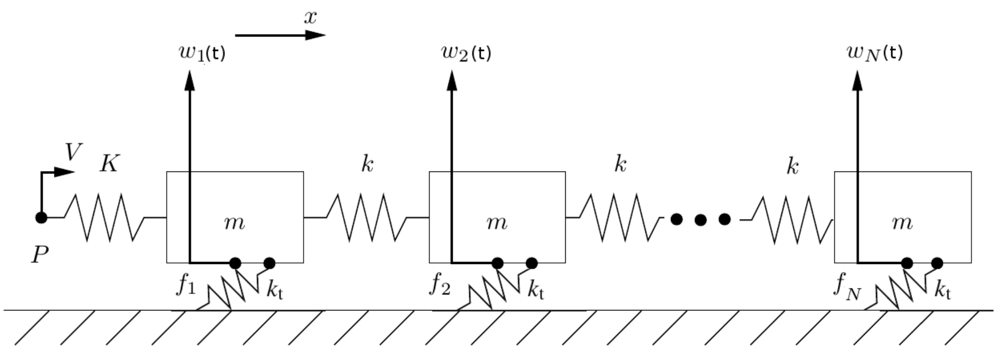

Figure 6.1: A sketch of the 1D Unloading model with time-dependent normal load

$$m\ddot{u}_n = \begin{cases} k(u_2 - u_1) + F_X(t) + F_1^\eta + f_1(t), & n = 1 \\ k(u_{n+1} - 2u_n + u_{n-1}) + F_n^\eta + f_n(t), & 2 \le n \le N-1 \\ k(u_{n-1} - u_N) + F_N^\eta + f_N(t), & n = N \end{cases} \tag{6.6}$$

It is almost the same as the one we had for the side-driven model except we now have a time-dependent friction force.
The connection between the surface and the slider is set up exactly the same as the one-dimensional model in chapter five.

6.2 The numerical model

When we initialize the system we set the initial normal load to be F_N^{start}. The initial shear load is zero, so we need to change this. The first two seconds of the simulation is used to do this. During this time F_N^{start} is held constant, and the point, P in Figure 6.1, starts to move with a constant velocity, V. Tension builds up in the spring connecting P and the first block, there is now a increasing shear load on the system. In Figure 6.3 we can see how the shear force builds up during the first 0.75s (In this case we have used 0.75s instead of 2s to better correspond with the experimental data). When $t = 0.75$s the velocity of P is set to zero, and the position of P is held constant throughout the whole simulation. Now we start to reduce the normal load from F_N^{start} to F_N^{end}. We have used a linear relationship between F_N^{start} to F_N^{end}. When the normal load reaches F_N^{end}, the simulation is ended.
In Appendix B.6 the code for the unloading model is attached.

6.3 The experimental data

To get the model working we used some results given to us by the group of S.D. Glaser at the University of California, Berkeley. In Figure 6.2, we can see the normal load and the shear load plotted against time. As we can see there are some strange behavior in this figure. First we can see that between 1000 and 1500s there is a sudden stopp in the reduction process of the normal load. The explanation is, at $t = 1000$s there was some event and the

observers wanted to examine the experiment. Just to make the numerical data and the experimental data look alike, we have also put in this "break". In the experiment a sudden slip will also cause the normal force to alter. This effect is not taken into the numerical model, so if we compare Figure 6.2 and Figure 6.3, we see that the "wavy" behavior of the experimental data is not in the numerical model.
Just around $t = 3000$s we can see some strange behavior in the experimental data. Here the shear force becomes greater than the normal force. We'll not try to explain why this happens, because we do not know exactly how the experiment where conducted, and therefore we do not have all the data required to do that. One way to achieve this effect is to have a quite large μ_s, and to be able to recreate the experimental data we had to use $\mu_s = 2.6$.
To get a full working model, we also have to estimate μ_d. This was done by using the experimental data in Figure 6.4. As we have seen from the results from the one-dimensional model, the normalized share force oscillates around μ_d after the first globals slip $L_P/L = 1$. If we use this knowledge on the experimental data in Figure 6.4 we can see that around $t = 3250$s we have a sudden drop in $F_X(t)/F_N$, this indicates that there has been a global slip. If we now subtract $(F_X(t)/F_N)_{\min,t=3250}$ from $(F_X(t)/F_N)_{\max,t=3250}$, we get an estimated $\mu_d = 1.2$.
If we now compare Figure 6.4 and Figure 6.5, we can see that there are some similarities and some differences. The most obvious differences is the difference between $(F_X(t)/F_N)_{\min}$ and $(F_X(t)/F_N)_{\max}$ in the two cases. We do not have any good explanation of this, but there is probably some factors we have not included in the numerical model that makes the difference. If we look at the shape of the curve $t = 2000$s to $t = 3250$s in the experimental results and the curve between $t = 2$s to $t = 3.2$s in the experimental results, we see that both the experimental data and the numerical results has the same shape. If we compare this to the one-dimensional model we see that the shape is non-linear.
The last data we got from the group was the displacement, Figure 6.6 of five measuring points. We have been told that there probably is something wrong with some of the sensors, which might explain the strange behavior of device four and five. We'll not use to much time on these data, but if we look at the total displacement, we can see, if we compare the experimental data with the numerical data Figure 6.7, that there is a quite big difference between the experimental data and the numerical data. The maximum displacement in the experimental data is ~ 0.0225mm. In the numerical data the number is close to 1.9mm which is close to 85 times higher than the experimental displacement. The factor that affect the displacement in the numerical model is the stiffness in the driving spring. With a small spring stiffness we need to stretch the spring a longer distance to achieve the same spring force as we would with a spring with a higher spring stiffness. By using this knowledge, we can, by tweaking the value of the driving spring stiffness, probably achieve to get about the same displacement in the numerical model as they did in the experiment. But because of short time, we have not been able to investigate this further.
We have chosen not to use the values for μ_s and μ_d obtained here. The μ_s and μ_d we got from comparing the experimental data and the numerical model is quite large compared to what we used in chapter five, and the values that have been used in the experiments

in chapter three. Therefore we'll use the same values $\mu_s = 0.7$ and $\mu_d = 0.45$, the same values we used in chapter five.

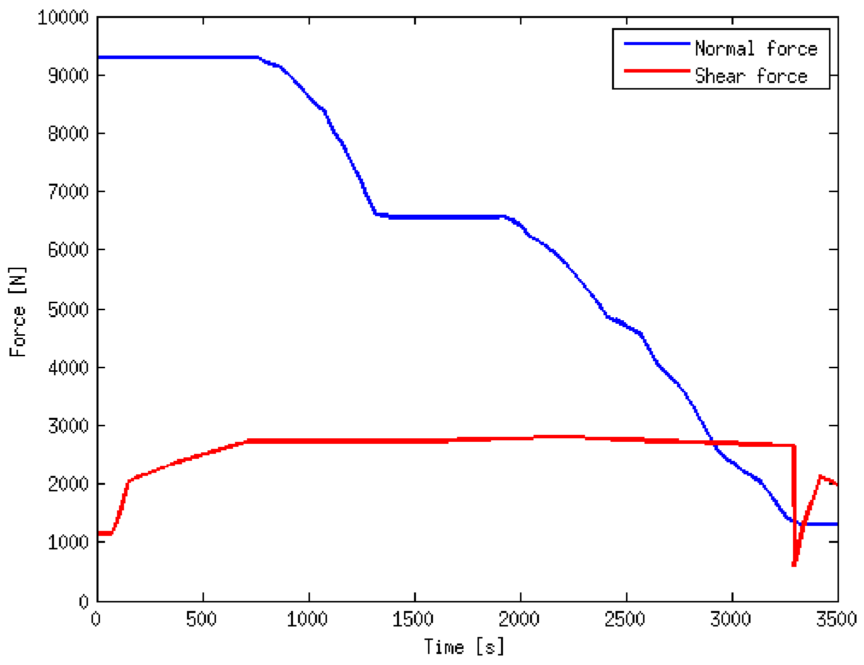

Figure 6.2: Experimental data of the normal force and the shear force from a unloading experiment. The red lower line is the shear force, and the blue upper line is the normal force.

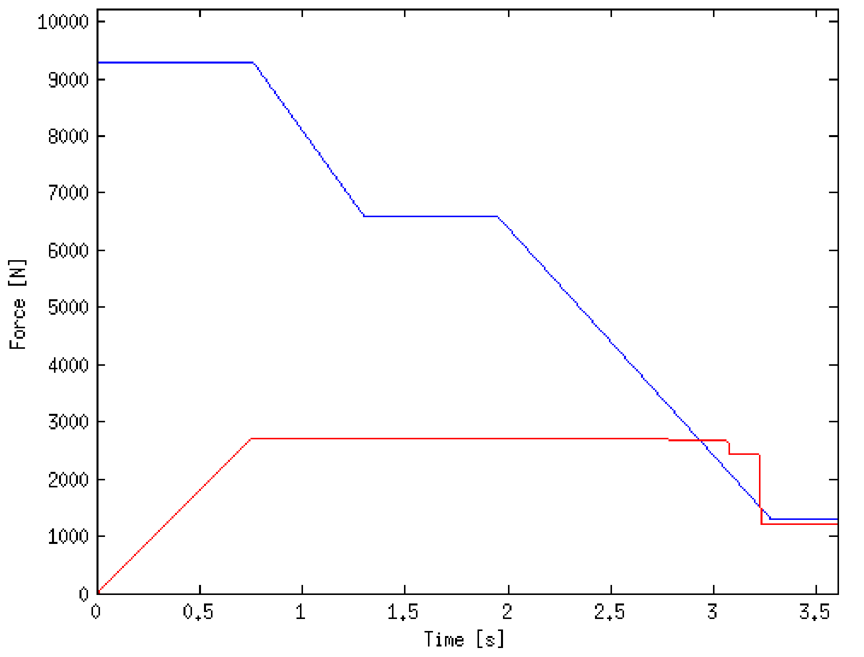

Figure 6.3: Numerical results of the normal force and the shear force from a unloading experiment. The red lower line is the shear force, and the blue upper line is the normal force. Here we have used $\mu_s = 2.6$, $\mu_d = 1.2$ and $N = 181$

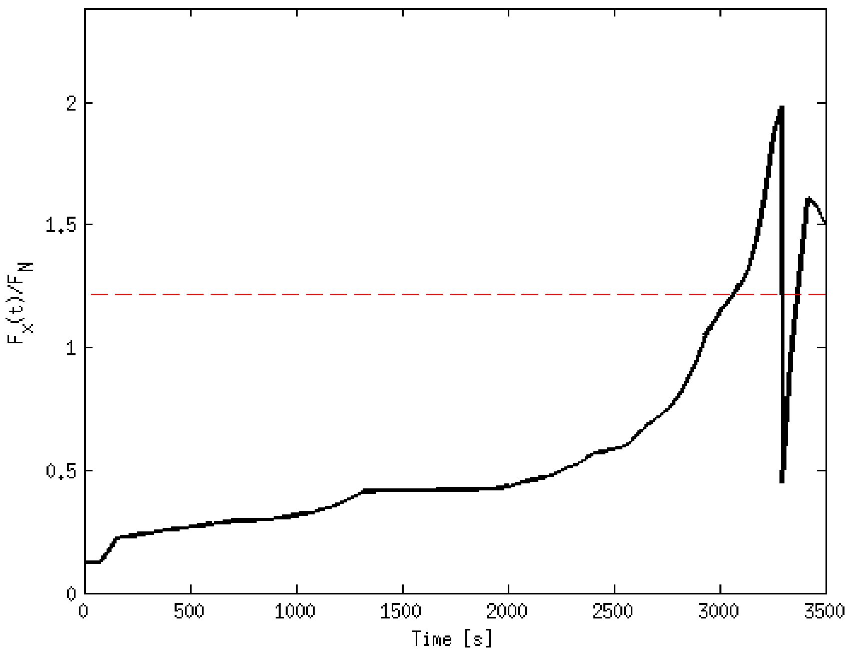

Figure 6.4: In this plot we have taken the data from Figure 6.2 and divided the shear force $F_X(t)$ with the normal force, F_N. The black curve is $F_X(t)/F_N$. The red dashed line is an estimate for the dynamic friction coefficient

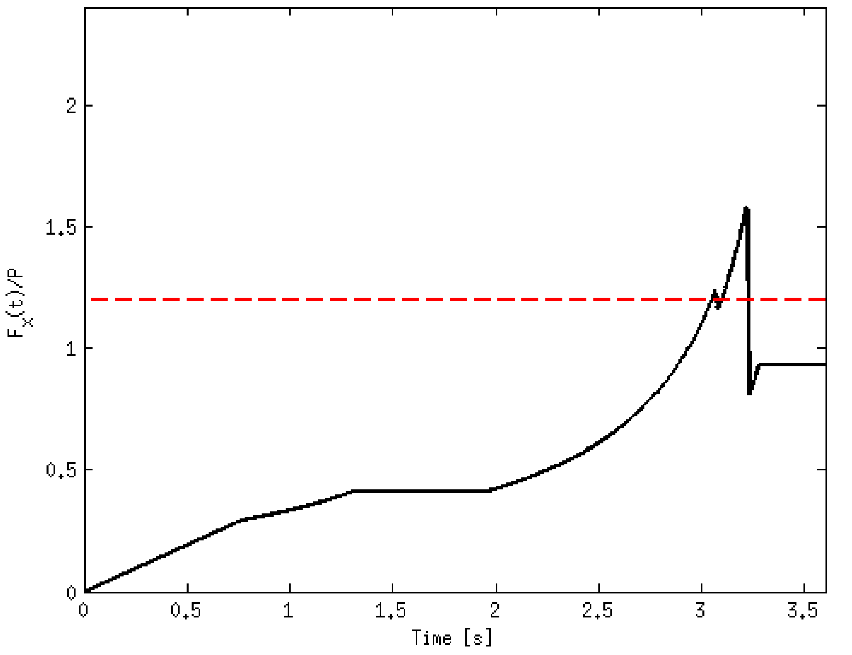

Figure 6.5: In this plot we have tried to make a plot similar to the plot in Figure 6.4. The black curve is $F_X(t)/F_N$. The red dashed line is the dynamic friction coefficient which is $\mu_d = 1.2$ and $N = 181$

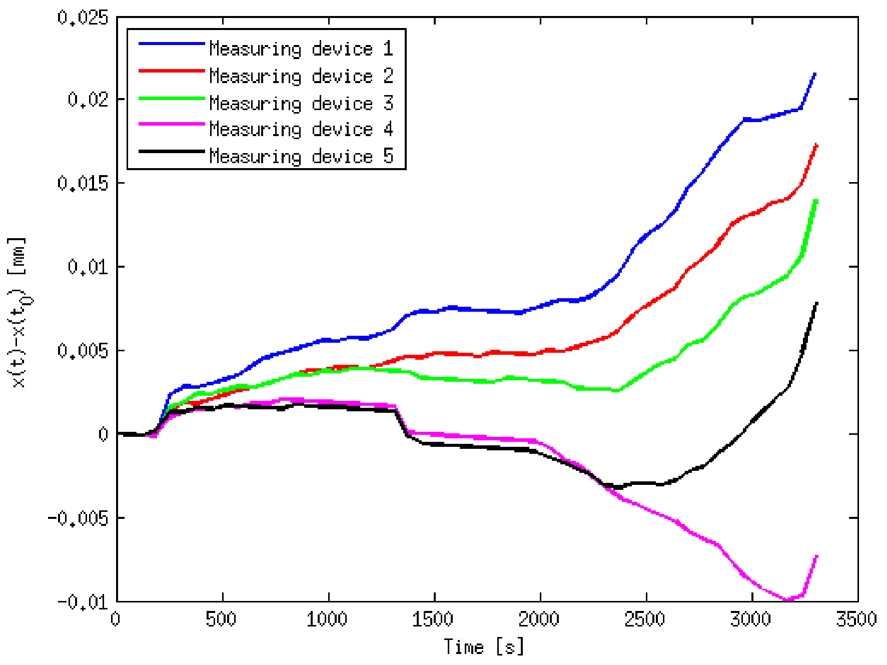

Figure 6.6: The plot show the displacement of five different points on the experimental slider.

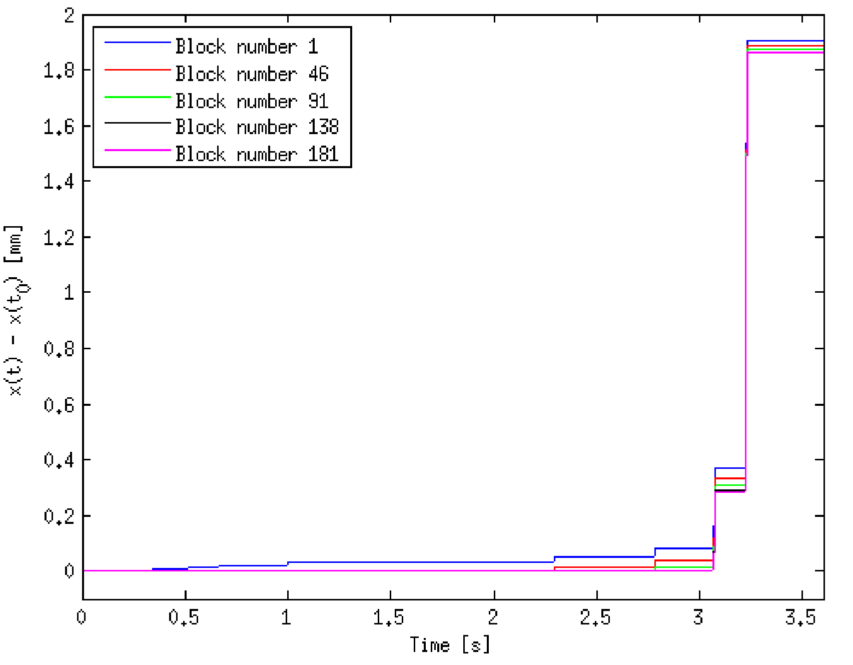

Figure 6.7: The plot show the displacement of five different blocks from the numerical simulation. $N = 181$

6.4 Results

In these experiments we have used $\mu_s = 0.7$ and $\mu_d = 0.45$. First we'll analyze the effect of different F_N^{start}. We have used the values $F_N^{\text{start}} = 400\text{N}$, $F_N^{\text{start}} = 600\text{N}$ and $F_N^{\text{start}} = 800\text{N}$.
If we first just look at the shape of the loading curves in Figure 6.8, 6.9, 6.10, 6.11 and 6.12 and compare them to the loading curves from the one-dimensional model, we can see that the shape of the "sawtooth" if quite different. In the case of the one-dimensional model there was a linear connection between the bottom of the sawtooth and the top of the sawtooth. While in this case it seems to be some kind of curved line.
If we assume that for the one-dimensional model we had a connection on the linear form $y = ax + b$, we have in this case the integral of this, $y = \int(ax + b)dx$.
We also see that when we get a slip and the sawtooth reaches a maximum and drops down to a minimum the relationship between the max and min always seems to be $\mu_d \pm x$, where x is some number smaller than μ_s.
Before $t = 2\text{s}$, where t is the time the loading curve follows the same rules as the loading curve for the one-dimensional model. The explanation of this is that before $t = 2\text{s}$ the normal load is kept constant.
If we now take a look at when the first global slip occurs

1. For $F_N^{\text{start}} = 400\text{N}$ $t = 2.65\text{s}$
2. For $F_N^{\text{start}} = 600\text{N}$ $t = 3.8\text{s}$
3. For $F_N^{\text{start}} = 800\text{N}$ $t = 4.1\text{s}$
4. For $F_N^{\text{start}} = 400\text{N}$ and $\beta = 0.45$ $t = 2.8\text{s}$
5. For $F_N^{\text{start}} = 400\text{N}$ and $\beta = 0.225$ $t = 3.2\text{s}$

From this we see that if we increase F_N^{start}, the first global slip will come later. When it comes to the change in β the results is not very conclusive. We get the highest time for $\beta = 0.225$ which is the middle value of β. This might indicate that there is something wrong with the model, or that there is something we have overseen.

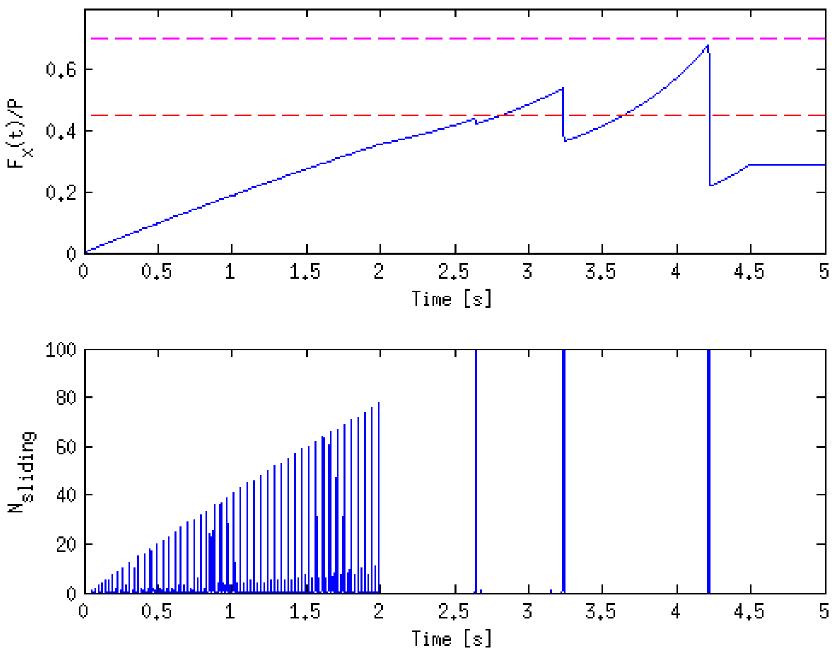

Figure 6.8: $F_X(t)/F_N$ for $F_N^{\text{start}} = 400\text{N}$, $F_N^{\text{end}} = 100\text{N}$, $\mu_s = 0.7$, $\mu_d = 0.45$. The pink dashed line is μ_s, the red dashed line is μ_d and the blue line is $F_X(t)/F_N$. Under the $F_X(t)/F_N$ plot we also have a plot of the number of blocks that are sliding.

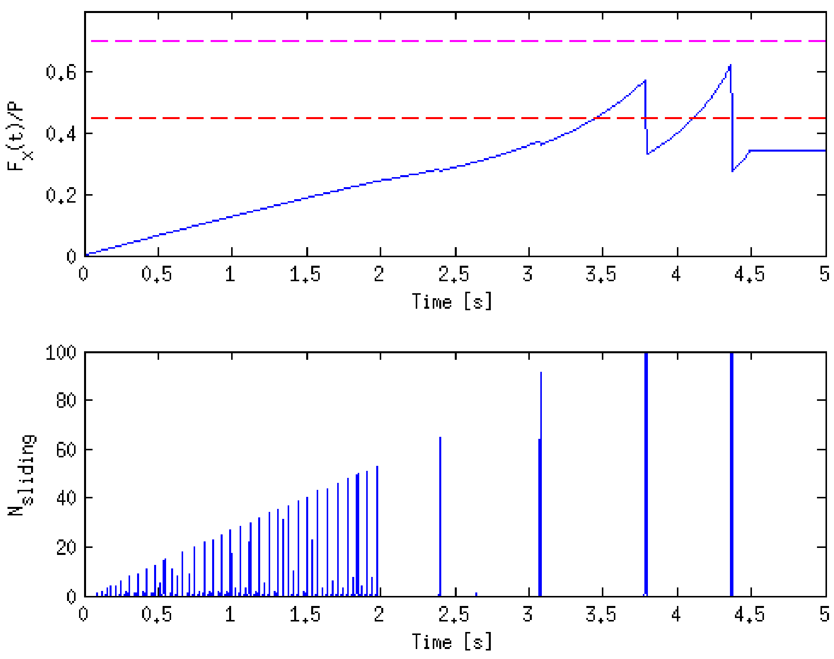

Figure 6.9: $F_X(t)/F_N$ for $F_N^{\text{start}} = 600\text{N}$, $F_N^{\text{end}} = 100\text{N}$, $\mu_s = 0.7$, $\mu_d = 0.45$. The pink dashed line is μ_s, the red dashed line is μ_d and the blue line is $F_X(t)/F_N$. Under the $F_X(t)/F_N$ plot we also have a plot of the number of blocks that are sliding.

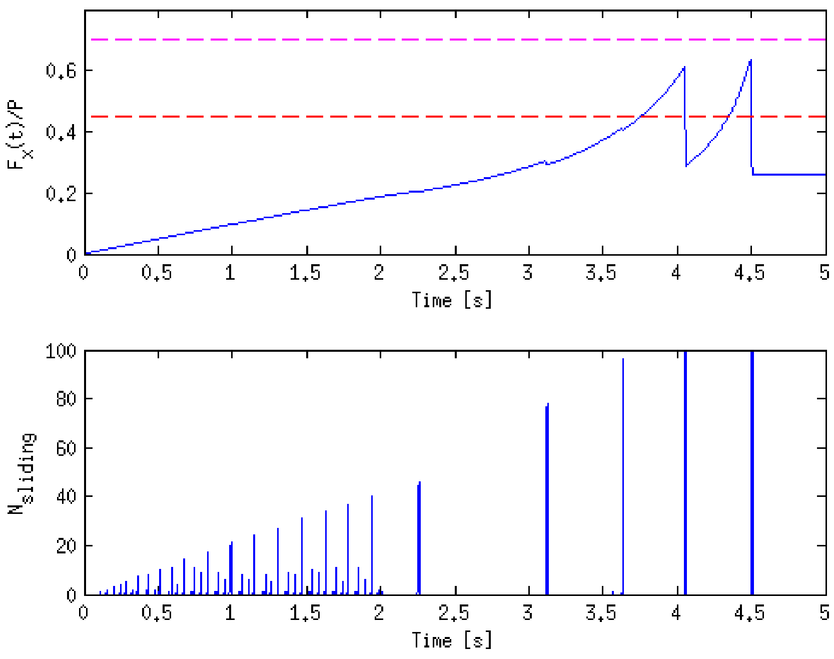

Figure 6.10: $F_X(t)/F_N$ for $F_N^{\text{start}} = 800\text{N}$, $F_N^{\text{end}} = 100\text{N}$, $\mu_s = 0.7$, $\mu_d = 0.45$. The pink dashed line is μ_s, the red dashed line is μ_d and the blue line is $F_X(t)/F_N$. Under the $F_X(t)/F_N$ plot we also have a plot of the number of blocks that are sliding.

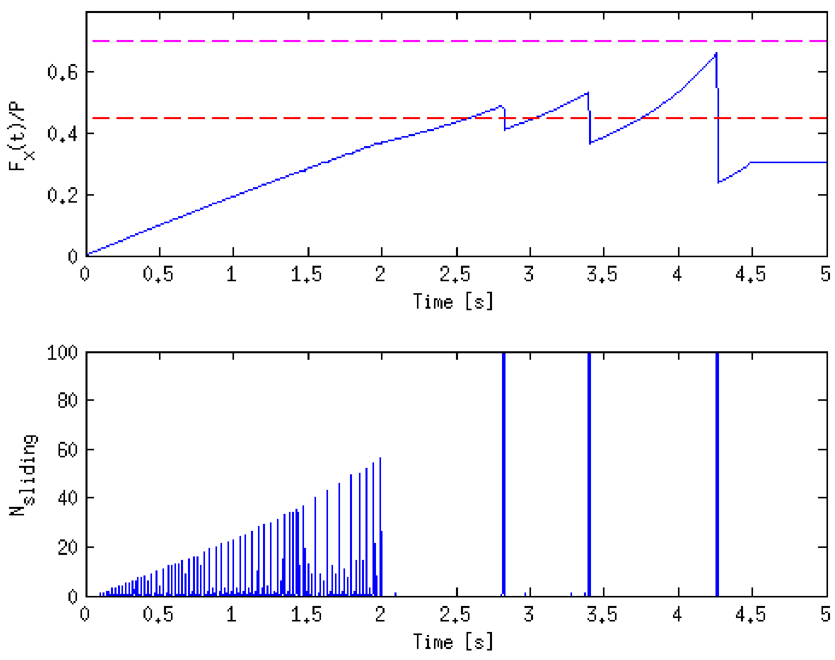

Figure 6.11: $F_X(t)/F_N$ for $F_N^{\text{start}} = 400\text{N}$, $F_N^{\text{end}} = 100\text{N}$, $\mu_s = 0.7$, $\mu_d = 0.45$ and $\beta = 0.45$. The pink dashed line is μ_s, the red dashed line is μ_d and the blue line is $F_X(t)/F_N$. Under the $F_X(t)/F_N$ plot we also have a plot of the number of blocks that are sliding.

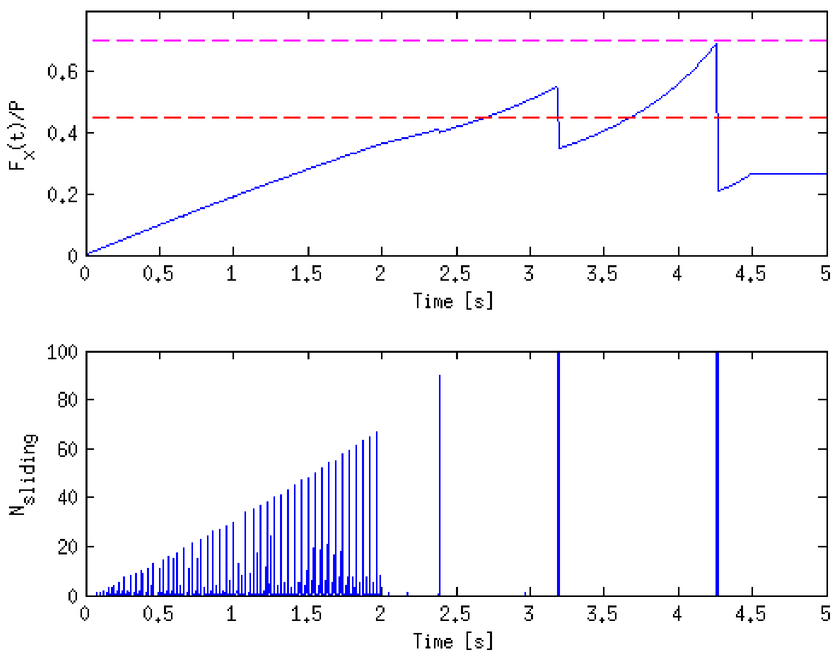

Figure 6.12: $F_X(t)/F_N$ for $F_N^{\text{start}} = 400\text{N}$, $F_N^{\text{end}} = 100\text{N}$, $\mu_s = 0.7$, $\mu_d = 0.45$ and $\beta = 0.225$. The pink dashed line is μ_s, the red dashed line is μ_d and the blue line is $F_X(t)/F_N$. Under the $F_X(t)/F_N$ plot we also have a plot of the number of blocks that are sliding.

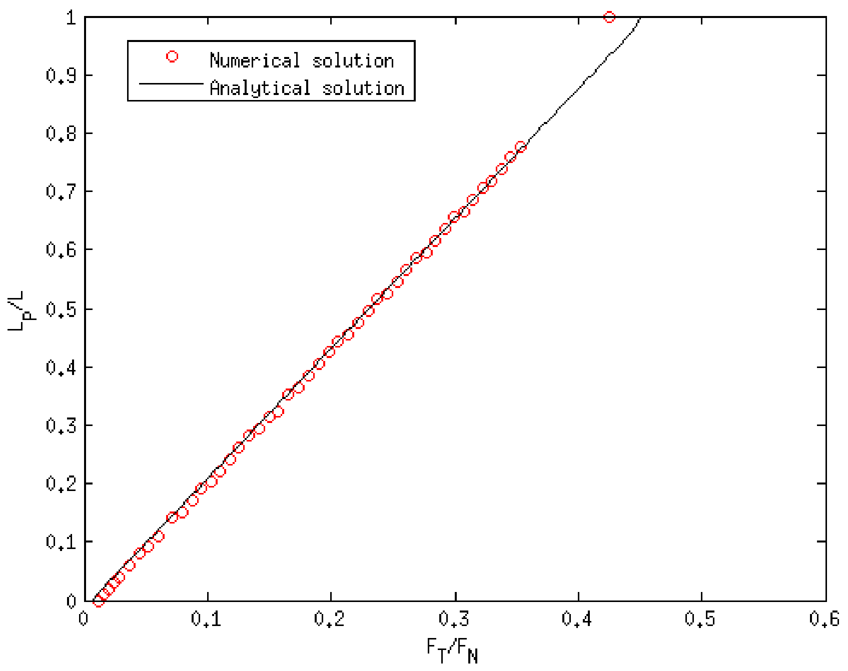

Figure 6.13: $L_p/L - F_T/F_N$ for $F_N^{\text{start}} = 400\text{N}$, $F_N^{\text{end}} = 100\text{N}$, $\mu_s = 0.7$, $\mu_d = 0.45$. The analytical solution is not valid for the data after $T_U^{\text{start}} = 2\text{s}$

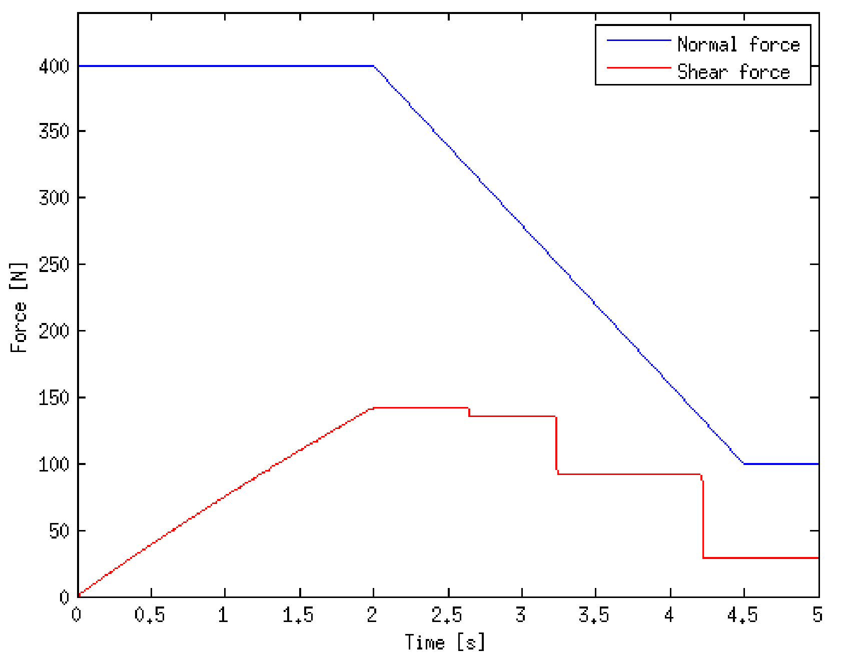

Figure 6.14: Numerical results of the normal force and the shear force from a unloading experiment. The red lower line is the shear force, and the blue upper line is the normal force. Here we have used $\mu_s = 0.7$, $\mu_d = 0.45$, $F_N^{\text{start}} = 400\text{N}$, $F_N^{\text{end}} = 100\text{N}$ and $N = 100$

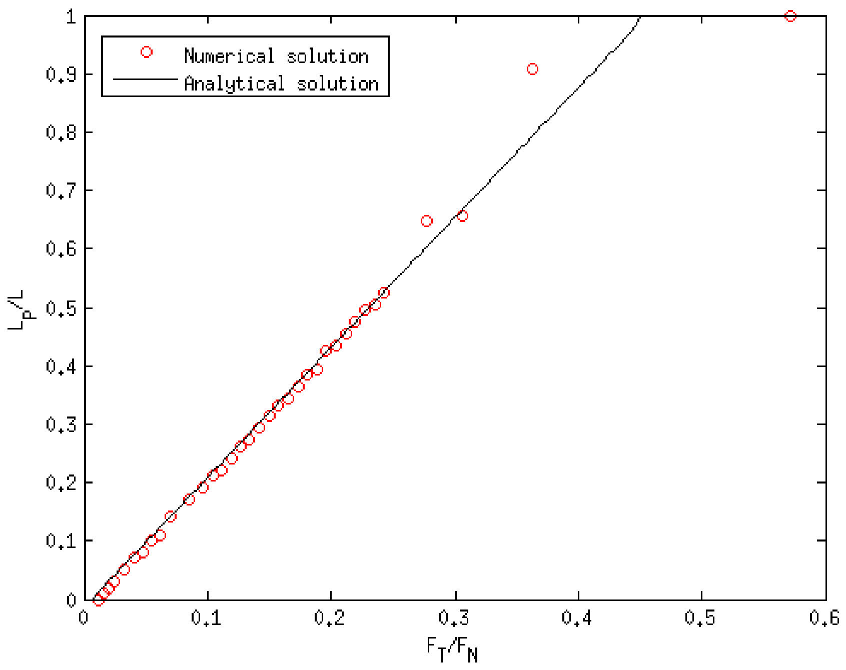

Figure 6.15: $L_p/L - F_T/F_N$ for $F_N^{\text{start}} = 600\text{N}$, $F_N^{\text{end}} = 100\text{N}$, $\mu_s = 0.7$, $\mu_d = 0.45$. The analytical solution is not valid for the data after $T_U^{\text{start}} = 2\text{s}$

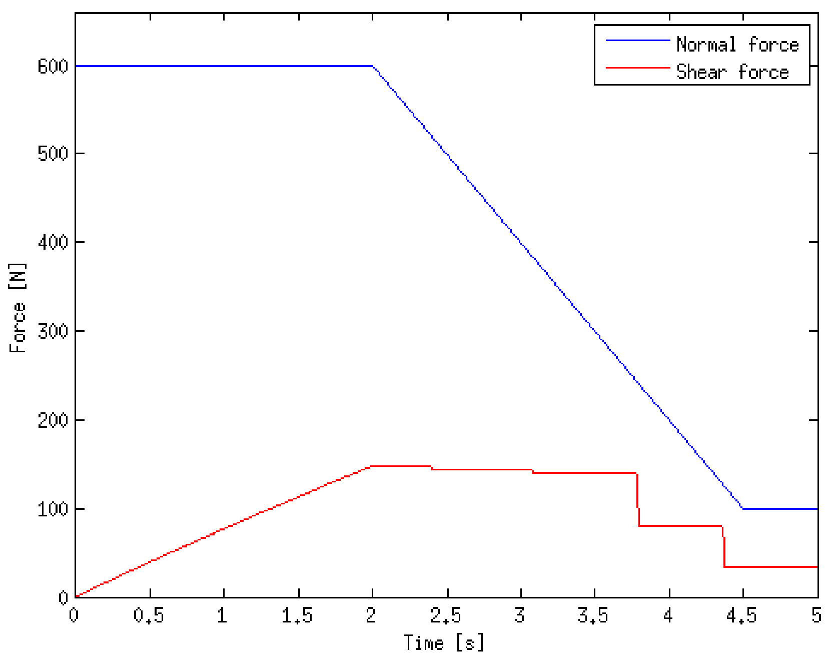

Figure 6.16: Numerical results of the normal force and the shear force from a unloading experiment. The red lower line is the shear force, and the blue upper line is the normal force. Here we have used $\mu_s = 0.7$, $\mu_d = 0.45$, $F_N^{\text{start}} = 600\text{N}$, $F_N^{\text{end}} = 100\text{N}$ and $N = 100$

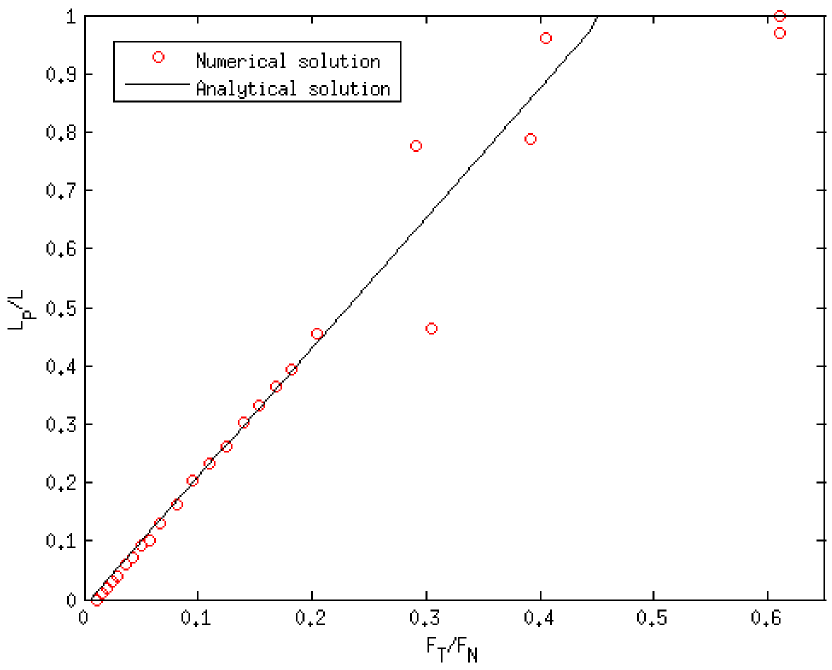

Figure 6.17: $L_p/L - F_T/F_N$ for $F_N^{\text{start}} = 800\text{N}$, $F_N^{\text{end}} = 100\text{N}$, $\mu_s = 0.7$, $\mu_d = 0.45$. The analytical solution is not valid for the data after $T_U^{\text{start}} = 2\text{s}$

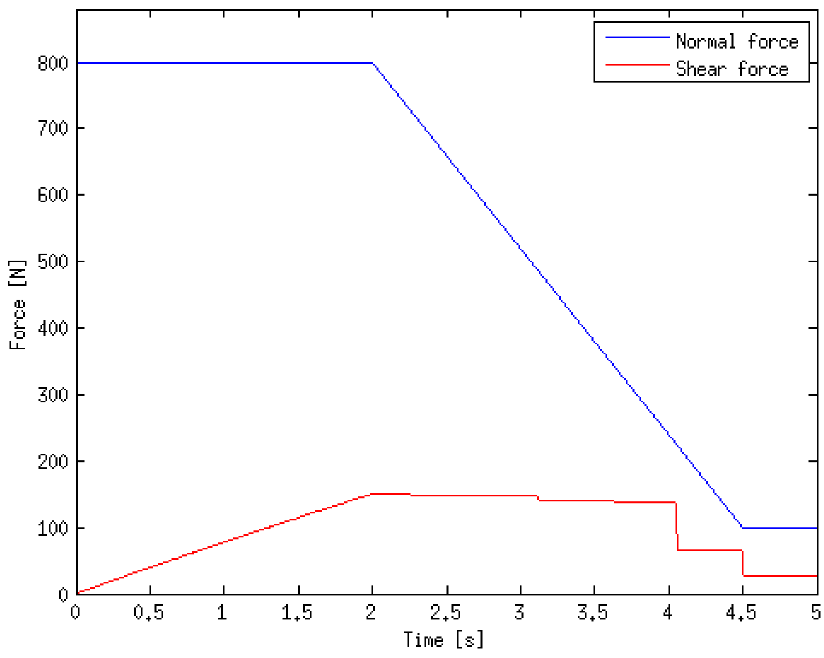

Figure 6.18: Numerical results of the normal force and the shear force from a unloading experiment. The red lower line is the shear force, and the blue upper line is the normal force. Here we have used $\mu_s = 0.7$, $\mu_d = 0.45$, $F_N^{\text{start}} = 800\text{N}$, $F_N^{\text{end}} = 100\text{N}$ and $N = 100$

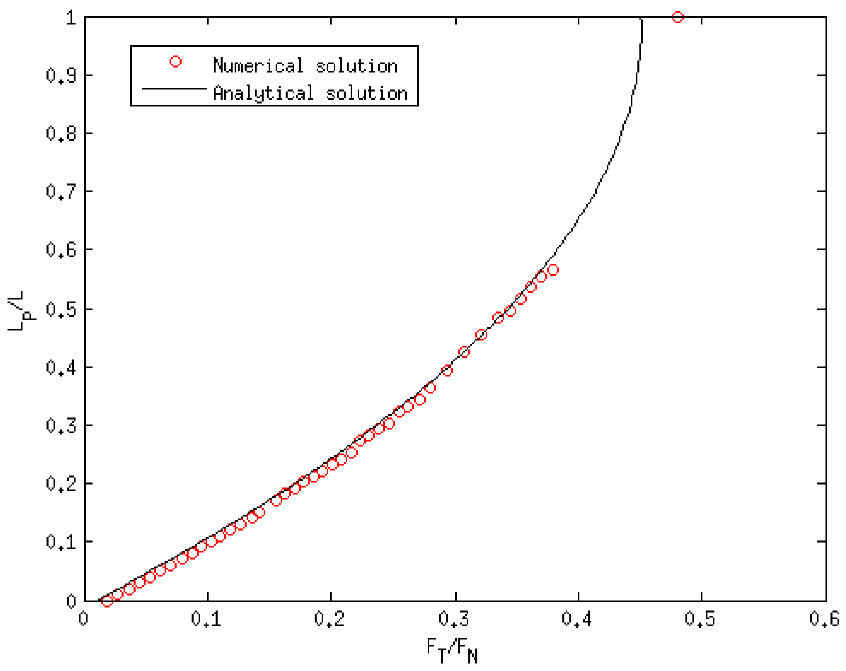

Figure 6.19: $L_p/L - F_T/F_N$ for $F_N^{\text{start}} = 400\text{N}$, $F_N^{\text{end}} = 100\text{N}$, $\mu_s = 0.7$, $\mu_d = 0.45$ and $\beta = 0.45$. The analytical solution is not valid for the data after $T_U^{\text{start}} = 2\text{s}$

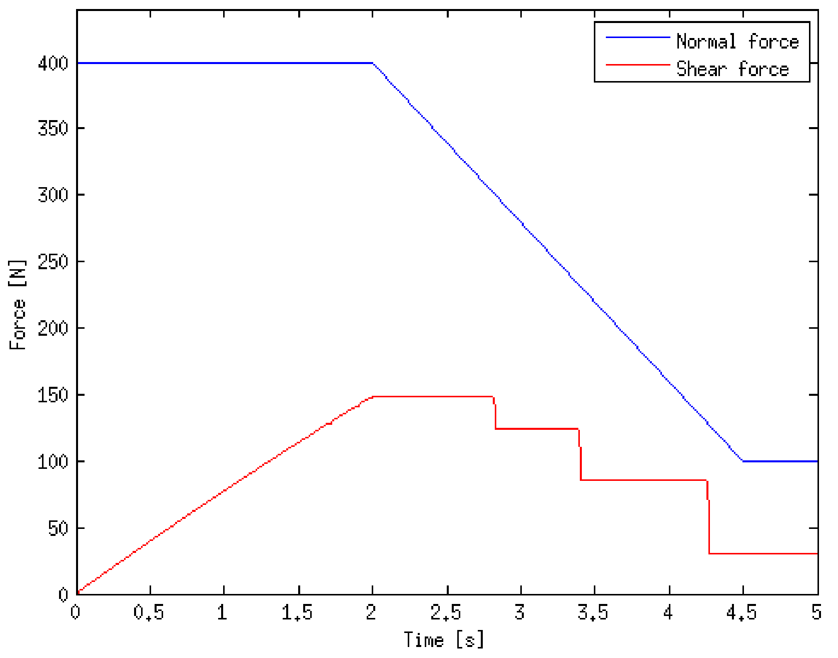

Figure 6.20: Numerical results of the normal force and the shear force from a unloading experiment. The red lower line is the shear force, and the blue upper line is the normal force. Here we have used $\mu_s = 0.7$, $\mu_d = 0.45$, $F_N^{\text{start}} = 400\text{N}$, $F_N^{\text{end}} = 100\text{N}$, $N = 100$ and $\beta = 0.45$

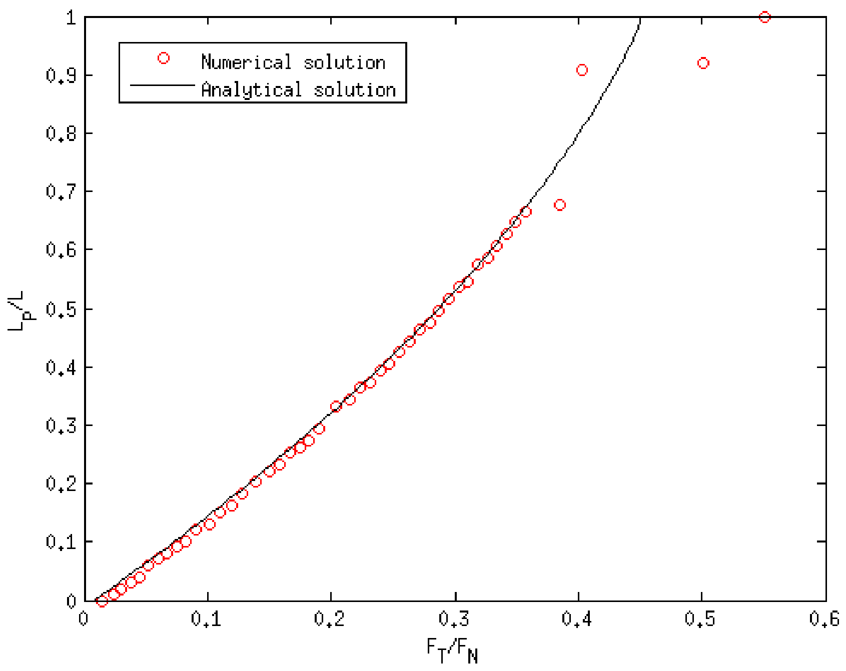

Figure 6.21: $L_p/L - F_T/F_N$ for $F_N^{\text{start}} = 400\text{N}$, $F_N^{\text{end}} = 100\text{N}$, $\mu_s = 0.7$, $\mu_d = 0.45$ and $\beta = 0.225$. The analytical solution is not valid for the data after $T_U^{\text{start}} = 2\text{s}$

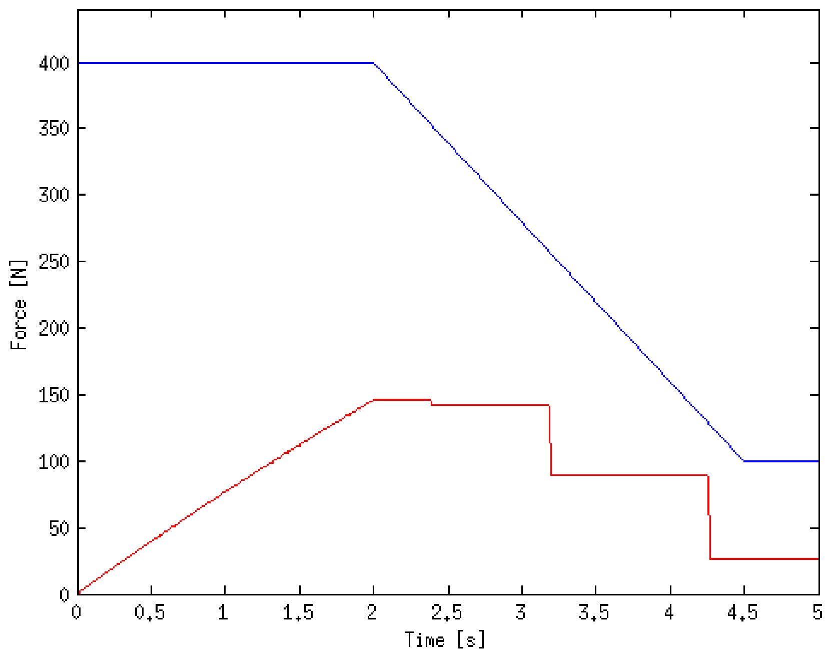

Figure 6.22: Numerical results of the normal force and the shear force from a unloading experiment. The red lower line is the shear force, and the blue upper line is the normal force. Here we have used $\mu_s = 0.7$, $\mu_d = 0.45$, $F_N^{\text{start}} = 400\text{N}$, $F_N^{\text{end}} = 100\text{N}$, $N = 100$ and $\beta = 0.225$

Part VII

Discussion

Chapter 7

Discussion

7.1 Side driven model

In chapter five we introduced the one-dimensional side-driven model. We derived some analytical solution both for a skewed normal load, where we used the values $\theta = -0.8333$, $\theta = 0$ and $\theta = 0.8333$ to describe the skewness of the normal load. We also derived an analytical solution for the initial shear force, where we used the values $\beta = 0.45$, $\beta = 0.225$ and $\beta = 0$ to describe the distribution of force between the nodes.
We used the above values to see how they affected the precursor length in the simulations. In Figure 5.9 we could see that the precursor length would "grow" faster for low values of F_T/F_N when the normal force was low at the driving side (negative values of θ). And that it would grow slowe for low values of F_T/F_N when the normal force was high at the driving side (positive values of θ).
In Figure 5.12 we could see the value of β affected the shape of the L_P/L-F_T/F_N-curve. With a $\beta > 0$ the first node gets larger initial spring force directed in the opposite direction as the driving force, and as we can se from the figure, the "growth"-rate of the precursor length is smaller for small F_T/F_N.
We also saw how the variation of μ_s and μ_d affected the system. We concluded that the cases we looked at only depended on the dynamic friction coefficient.

7.2 Top driven model

We did not get enough time to explore this model. We only got time to write the code and do some trial-runs of this code.

7.3 Unloading model

Here we used some of the experimental data given to us at the group at the University of California, Berkeley, to test our model. We saw that the model had a very good behavior compared to the experimantal data. We got time to do some test runs of the code, and tried some different values for F_N^{start} and got some result from that.
The numerical model is not working that well. A lot of the simulation-time is spent by initiating the shear load (the first 2 seconds). This should probably be done in an other and better way. Because it is a linear connection between the value of the shear load at

$t = 0$s and $t = 2$s, we can probably calculate the behavior of the system analytical for this time-interval. In the master thesis of J.Trømborg [27] they did this for the one-dimensional model.

7.4 Conclusion

Compared to the experimental data from Maegawa et al. the one-dimensional model does not behave that good. As we saw in chapter five, the gap between the experimental data and the numerical model was quite large. To better represent the reality we need more dimensions in our numerical model. J.Trømborg looked at a (1+1d) model in his thesis [27]. This model gives an extra dimension normal to the plane. This leads to a better model between the normal load and the surface because the normal load is connected through springs all the way down to the surface. In the master thesis of D.S. Amundsen [2] he conducted some numerical simulations involving two dimensions in the plane, a (2+0D) model. This model gives us better insight into what happens in the contact area. In the one-dimensional model we have to do many tricks to get the model to behave more like the real world. One of the trick we have used is to include the initial shear force. In the (1+1d) model we do not need this, because the connections between the nodes in the extra dimension takes care of this effect. The one-dimensional model is a very good tool to use if it is used correctly, and the user knows that it has some disadvantages. The one-dimensional model is also very good to use as a “discovery“-model, a model we can use to build up the knowledge about the problem we are working with. To get the full advantage of the (1+1d)-model and the (2+0D)-model we need to make a three-dimensional model. The big problem with the three-dimensional model is the number of nodes, N^3, we have to simulate. In the one-dimensional model we only have to calculate two springs for every node. In the two-dimensional model we have to calculate 8, and in the three-dimensional model we have to calculate 26 springs for each node.

7.5 Future work

The side-driven one-dimensional model has a weakness when it comes to the application of the shear force. If the shear force is going to reach the last node in this model it has to work trough all the other nodes. In chapter five we saw how we could manipulate the relationship between the friction spring stiffness and the spring stiffness between the nodes to make the shear fore affect more of the nodes. With a top-driven model we can transfer the shear load to all the nodes in the system. If we divide the shear load uniformly on all the noes all the node will reach the static friction threshold at the same time and we’ll get one gigantic global slip. So in a future work top-driven model we should try to come up with a model for the shear force distribution that mimic that of the (1+1d)-model.
When it comes to the unloading model a lot of work has to be done. When we load the model with a high normal load it is probably important to include the fact that the block will be pressed together in the direction of the normal load, and expand in the direction normal on the normal load.

7.6 Concluding words

When I now look back at the time spent doing this master thesis I realize that too much of the time have been spent writing the code. What I have learned from this experience is that, all time spent writing unused code is a waste of time.

Appendix A

Viscous Damping

A.1 Strange behavior of the viscous damping

In Figure A.1 and A.2 we can see two different plots of the velocity of the five first blocks. The plots are from two differen simulations where only the value of η has been changed. We can also see how many blocks that are actually sliding. In Figure A.1, where $\eta = \sqrt{0.1}$, we can see that we have an over-damped system and the blocks continue sliding because they always will have a positive velocity. This is not desirable. In Figure A.2 we have changed the damping factor to $\eta = \sqrt{0.01}$. As we can see the system has become under-damped and the velocity of the block will reach a value that is zero or lower, and the blocks will go back to being under static friction.

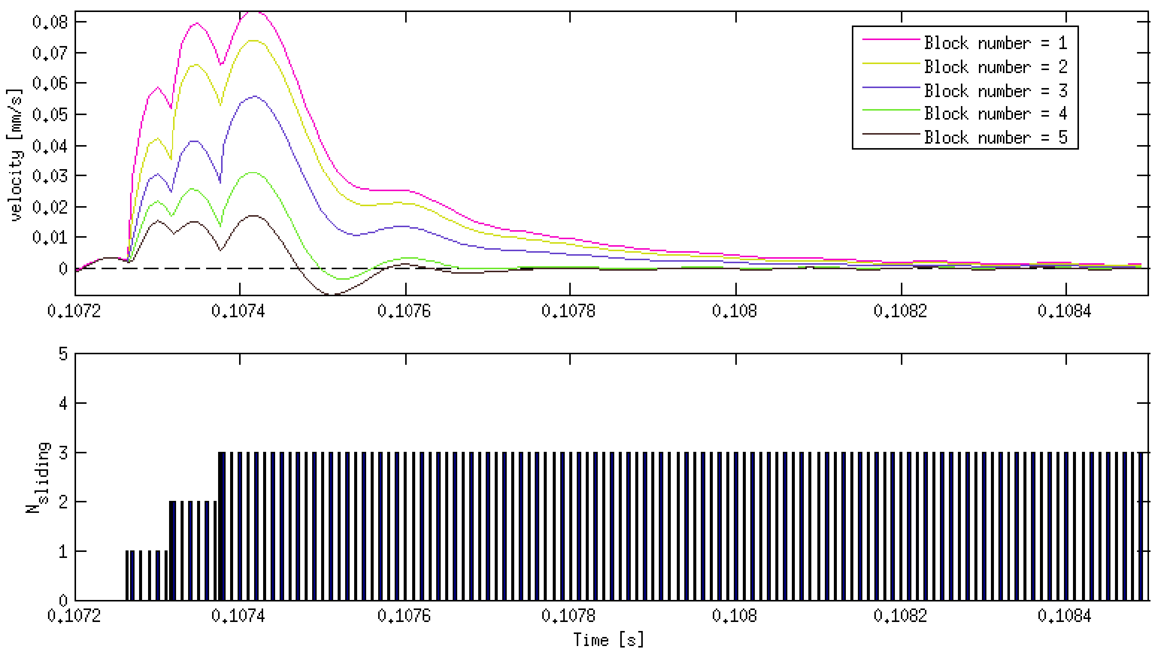

Figure A.1: This plot show the velocity for the five first blocks. When using $\eta = \sqrt{0.1}$ we get an over-damped system

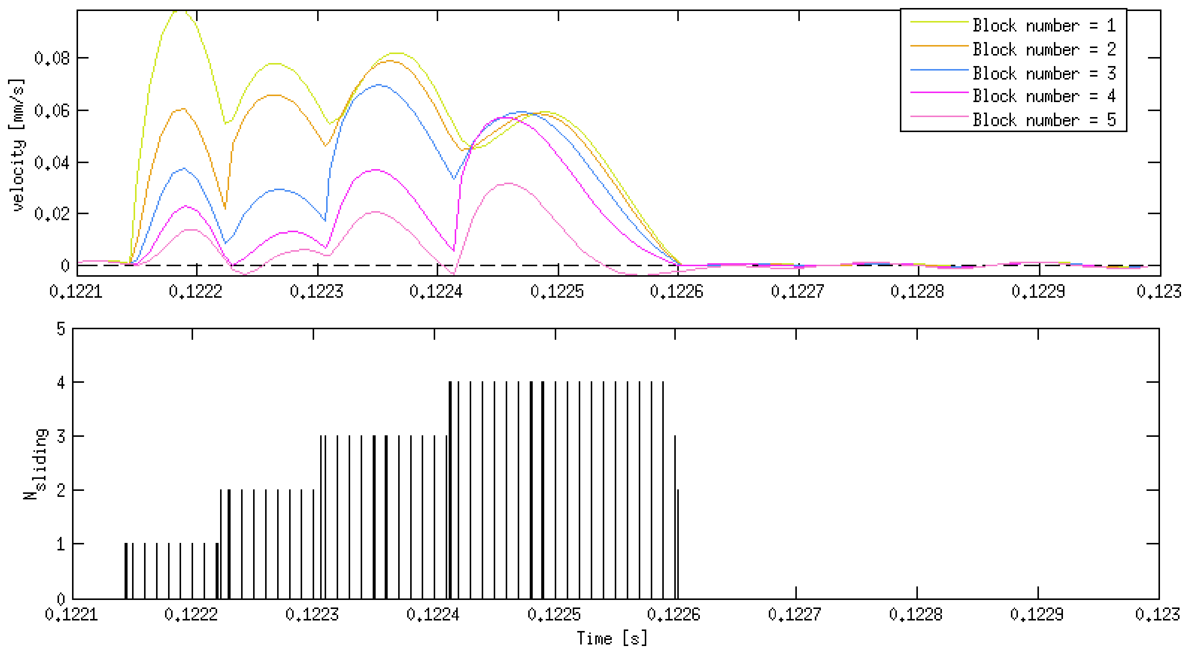

Figure A.2: This plot show the velocity for the five first blocks. When using $\eta = \sqrt{0.01}$ we get an under-damped system

Appendix B

C Code

B.1 Language

The code used in this thesis combines matlab and C. All the interface to set up a simulation and writing results to file is done with Matlab. The heavy number crunching is done in C. To combine these two languages an extension of Matlab, called MEX, has been used. By doing this we have tried to combine Matlabs easy-to-implement filewriting and plotting of results, and great number crunching properties of C's

B.2 Matlab code

B.2.1 Initialisation code

The following matlabcode creates a system of folders and creates the the files needed to store the data from the simulations

```
clear all;
clc
%--------------------------------------------------------------------------

%  The initial conditions
%--------------------------------------------------------------------------

%  T   : The total time of the simulation              [s]
%  dt  : Length of a time step
%  N   : Number of nodes
%  Lx  : The length of the block in the x-direction    [mm]
%  Ly  : The length of the block in the y-direction    [mm]
%  Lz  : The height of the block in the z-direction    [mm]
%  M   : Total block mass                              [kg]
%  E   : Young's modulus                               [GPa]
%  ny  : Relative viscous damping                      [sqrt(km)]
%  K   : Total driving spring stiffness                [MN/m]
%  Kn  : Driving spring stiffness per node             [MN/m]
%  Fz  : Applied normal load                           [N]
%  V   : Driving point velocity                        [mm/s]
%  mys : Static friction coefficient
%  myd : Dynamic friction coefficient
%  mysf: Static friction stiffness factor
%--------------------------------------------------------------------------

multiplier                              = 1;
run_time                                = 5;%3.6*multiplier;
dt                                      = 1E-8;
number_of_blocks                        = 100;
Lx                                      = 100;%181;%
Ly                                      = 5;%17;%
Lz                                      = 20;%60;%
total_system_mass                       = 0.012;
youngs_modulus                          = 2.5;
relative_viscous_damping                = sqrt(0.01);%0;%sqrt(0.01);%
str_eta                                 = 'eta-sqrt-0_01_';
driving_spring_stiffness                = 0.8;%8.0;%0.08;%
total_normal_force                      = 800;%200;400;%600;%
driving_velocity                        = 0.1/multiplier;
str_v                                   = 'V-0_1_';%'';%
static_friction_coefficient             = 0.7;%0.35;%1.4;%2.3;%
str_static_friction_coefficient         = '70';
delta_static_friction_coefficient       = 0;
dynamic_friction_coefficient            =
    0.45;%0.55;%0.65;%0.225;%0.9;%0.25;%0.35;%
```

```
str_dynamic_friction_coefficient          = '45';%'25_5';%'90';%'65';%'
   90';%'25';%'35';%
delta_dynamic_friction_coefficient        = 0;
theta                                     =
   0.0;%0.3333;%0.8333;%-0.1;%0.1333;%-0.1333;%-0.8333;%
static_friction_spring_stiffness_scaling = 1.0;%0.1;%0.5;%2.0;%10;%
str_sfsss                                 = '';%'sfsss -10_0_';%
beta                                      =
   0.45;%0;%0.225;%-0.45;%-0.225;%
str_beta                                  = 'beta-0_45_';%0.0;%'';%
gamma                                     = 0;
str_gamma                                 = 'gamma-profile_';%'gamma-
   neg-40_';%'';%
driving_first_node                        = 1;
driving_last_node                         = 0;

% print_node             = floor(number_of_blocks/2);
% print_freq             = ceil(10000*(1E-8/dt));
% print_counter_limit    = ceil(10000*(1E8*dt));
% print_counter_2_limit = ceil(100*(1E8*dt));

print_node            = floor(number_of_blocks/2);
print_freq            = ceil(1000*(1E-8/dt))*multiplier;
print_counter_limit   = ceil(1000*(1E8*dt))/multiplier;
print_counter_2_limit = ceil(10*(1E8*dt));
%----------------------------------------------------------------------------

%   Choose type of system
%----------------------------------------------------------------------------


alternative = 'avlastning';
% alternative = 'avlastning_with_ageing';
% alternative = 'avlastning_with_poisson';
% alternative = 'avlastning_desired_driving_force';
% alternative = 'avlastning_top_driven';
% alternative = 'avlastning_top_driven_with_poisson';
% alternative = 'side_driven';
% alternative = 'top_driven';
% alternative = 'left_right_driven';

%----------------------------------------------------------------------------

%   profile off; profile on; run_script; profile report;
%----------------------------------------------------------------------------

now_           = clock;
year           = now_(1);
month          = now_(2);
if (month < 10)
    month = sprintf('0%d',month);
```

```
else
    month = sprintf('%d',month);
end
day             = now_(3);
if (day < 10)
    day = sprintf('0%d',day);
else
    day = sprintf('%d',day);
end
hour            = now_(4);
if (hour < 10)
    hour = sprintf('0%d',hour);
else
    hour = sprintf('%d',hour);
end
minute          = now_(5);
if (minute < 10)
    minute = sprintf('0%d',minute);
else
    minute = sprintf('%d',minute);
end
second          = floor(now_(6));
if (second < 10)
    second = sprintf('0%d',second);
else
    second = sprintf('%d',second);
end

% Theta
if (theta > 0)
    str_theta = sprintf('pos-%d',abs(10000*theta));
elseif (theta < 0)
    str_theta = sprintf('neg-%d',abs(10000*theta));
else
    str_theta = sprintf('%d',theta);
end

% Static Friction Spring Stiffness Scaling
if (static_friction_spring_stiffness_scaling == 1)

elseif (static_friction_spring_stiffness_scaling > 1)
    str_sfsss = sprintf('sfsss-%d_',
        static_friction_spring_stiffness_scaling);
elseif (static_friction_spring_stiffness_scaling < 1)
    str_sfsss = sprintf('sfsss-0_%d_',10*
        static_friction_spring_stiffness_scaling);
end


% dt
if (dt == 1E-8)
```

```
    str_dt = '';
else
    str_dt = sprintf('_dt-%g',dt);
end

% Run Time
if (run_time > 1)
    str_run_time = sprintf('T-%d-%d_',floor(run_time), floor(10*(
        run_time - floor(run_time))));
else
    str_run_time = sprintf('T-0_%d_',10*run_time);
end


started         = datestr(now,'dd mmm yyyy, HH:MM:SS');
folder_root     = '~/Dropbox/Master/Amonton-Coulomb/1D_model';
switch alternative
    case 'side_driven'
        src_driven      = sprintf('%s/side_driven',folder_root);
        src_analyze     = sprintf('%s/analyze',src_driven);
        folder_driven   = sprintf('%s/results/runs/side_driven',
            folder_root);
        project_name    = sprintf('run_%d-%s-%s_%s-%s-%s_%sLx-%d_Ly-%
            d_Lz-%d_%sFN-%d_mu_s-%s_mu_d-%s_%s%stheta-%s_%s%sN-%d%s'
            ,...
                        year, month, day, hour, minute, second,...
                        str_run_time,Lx,Ly,Lz,...
                        str_v,total_normal_force,
                            str_static_friction_coefficient,
                            str_dynamic_friction_coefficient,...
                        str_sfsss,str_beta,str_theta,str_eta,
                            str_gamma,number_of_blocks,str_dt);
      alternative_short = 'SD';
        folder_project  = sprintf('%s/%s',folder_driven, project_name)
            ;
    case 'top_driven'
        src_driven      = sprintf('%s/top_driven',folder_root);
        src_analyze     = sprintf('%s/analyze',src_driven);
        folder_driven   = sprintf('%s/results/runs/top_driven',
            folder_root);
        project_name    = sprintf('run_%d-%s-%s_%s-%s-%s_%sLx-%d_Ly-%
            d_Lz-%d_%sFN-%d_mu_s-%s_mu_d-%s_%s%s_theta-%s_%s%sN-%d%s'
            ,...
                        year, month, day, hour, minute, second,...
                        str_run_time,Lx,Ly,Lz,...
                        str_v,total_normal_force,
                            str_static_friction_coefficient,
                            str_dynamic_friction_coefficient,...
                        str_sfsss,str_beta,str_theta,str_eta,
                            str_gamma,number_of_blocks,str_dt);
        alternative_short = 'TD';
        folder_project  = sprintf('%s/%s',folder_driven, project_name)
            ;
    case 'left_right_driven'
        src_driven      = sprintf('%s/left_right_driven',folder_root);
        src_analyze     = sprintf('%s/analyze',src_driven);
        folder_driven   = sprintf('%s/results/runs/left_right_driven',
            folder_root);
        project_name    = sprintf('run_%d-%s-%s_%s-%s-%s_%sLx-%d_Ly-%
            d_Lz-%d_%sFN-%d_mu_s-%s_mu_d-%s_%s%s_theta-%s_%s%sN-%d%s'
            ,...
                          year, month, day, hour, minute, second,...
                          str_run_time,Lx,Ly,Lz,...
                          str_v,total_normal_force,
                              str_static_friction_coefficient,
                              str_dynamic_friction_coefficient,...
                          str_sfsss,str_beta,str_theta,str_eta,
                              str_gamma,number_of_blocks,str_dt);
        folder_project  = sprintf('%s/%s',folder_driven, project_name)
            ;
        alternative_short = 'LRD';
    case 'avlastning'
        src_driven      = sprintf('%s/avlastning',folder_root);
        src_analyze     = sprintf('%s/analyze',src_driven);
        folder_driven   = sprintf('%s/results/runs/avlastning',
            folder_root);
        project_name    = sprintf('run_%d-%s-%s_%s-%s-%s_%sLx-%d_Ly-%
            d_Lz-%d_%sFN-%d_mu_s-%s_mu_d-%s_%s%stheta-%s_%s%sN-%d%s'
            ,...
                          year, month, day, hour, minute, second,...
                          str_run_time,Lx,Ly,Lz,...
                          str_v,total_normal_force,
                              str_static_friction_coefficient,
                              str_dynamic_friction_coefficient,...
                          str_sfsss,str_beta,str_theta,str_eta,
                              str_gamma,number_of_blocks,str_dt);
        folder_project  = sprintf('%s/%s',folder_driven, project_name)
            ;
        alternative_short = 'AL';
    case 'avlastning_desired_driving_force'
        src_driven      = sprintf('%s/avlastning_desired_driving_force
            ',folder_root);
        src_analyze     = sprintf('%s/analyze',src_driven);
        folder_driven   = sprintf('%s/results/runs/
            avlastning_desired_driving_force',folder_root);
        project_name    = sprintf('run_%d-%s-%s_%s-%s-%s_%sLx-%d_Ly-%
            d_Lz-%d_%sFN-%d_mu_s-%s_mu_d-%s_%s%s_theta-%s_%s%sN-%d%s'
            ,...
                          year, month, day, hour, minute, second,...
                          str_run_time,Lx,Ly,Lz,...
                          str_v,total_normal_force,
                              str_static_friction_coefficient,
                              str_dynamic_friction_coefficient ,...
                           str_sfsss ,str_beta ,str_theta ,str_eta ,
                              str_gamma ,number_of_blocks ,str_dt);
        folder_project  = sprintf('%s/%s',folder_driven, project_name)
           ;
        alternative_short = 'ALDDF';
    case 'avlastning_top_driven'
        src_driven      = sprintf('%s/avlastning_top_driven',
           folder_root);
        src_analyze     = sprintf('%s/analyze',src_driven);
        folder_driven   = sprintf('%s/results/runs/
           avlastning_top_driven',folder_root);
        project_name    = sprintf('run_%d-%s-%s_%s-%s-%s_%sLx-%d_Ly-%
           d_Lz-%d_%sFN-%d_mu_s-%s_mu_d-%s_%s%s_theta-%s_%s%sN-%d%s'
           ,...
                           year, month, day, hour, minute, second,...
                           str_run_time,Lx,Ly,Lz,...
                           str_v,total_normal_force,
                              str_static_friction_coefficient,
                              str_dynamic_friction_coefficient,...
                           str_sfsss,str_beta,str_theta,str_eta,
                              str_gamma,number_of_blocks,str_dt);
        folder_project  = sprintf('%s/%s',folder_driven, project_name)
           ;
        alternative_short = 'ALTD';
    case 'avlastning_top_driven_with_poisson'
        src_driven      = sprintf('%s/
           avlastning_top_driven_with_poisson',folder_root);
        src_analyze     = sprintf('%s/analyze',src_driven);
        folder_driven   = sprintf('%s/results/runs/
           avlastning_top_driven_with_poisson',folder_root);
        project_name    = sprintf('run_%d-%s-%s_%s-%s-%s_%sLx-%d_Ly-%
           d_Lz-%d_%sFN-%d_mu_s-%s_mu_d-%s_%s%s_theta-%s_%s%sN-%d%s'
           ,...
                           year, month, day, hour, minute, second,...
                           str_run_time,Lx,Ly,Lz,...
                           str_v,total_normal_force,
                              str_static_friction_coefficient,
                              str_dynamic_friction_coefficient,...
                           str_sfsss,str_beta,str_theta,str_eta,
                              str_gamma,number_of_blocks,str_dt);
        folder_project  = sprintf('%s/%s',folder_driven, project_name)
           ;
        alternative_short = 'ALTDWP';
    case 'avlastning_with_poisson'
        src_driven      = sprintf('%s/avlastning_with_poisson',
           folder_root);
        src_analyze     = sprintf('%s/analyze',src_driven);
        folder_driven   = sprintf('%s/results/runs/
           avlastning_with_poisson',folder_root);
        project_name    = sprintf('run_%d-%s-%s_%s-%s-%s_%sLx-%d_Ly-%
            d_Lz-%d_%sFN-%d_mu_s-%s_mu_d-%s_%s%s_theta-%s_%s%sN-%d%s'
            ,...
                           year, month, day, hour, minute, second,...
                           str_run_time,Lx,Ly,Lz,...
                           str_v,total_normal_force,
                              str_static_friction_coefficient,
                              str_dynamic_friction_coefficient,...
                           str_sfsss,str_beta,str_theta,str_eta,
                              str_gamma,number_of_blocks,str_dt);
        folder_project  = sprintf('%s/%s',folder_driven, project_name)
            ;
        alternative_short = 'ALWP';
    case 'avlastning_with_ageing'
        src_driven      = sprintf('%s/avlastning_with_ageing',
            folder_root);
        src_analyze     = sprintf('%s/analyze',src_driven);
        folder_driven   = sprintf('%s/results/runs/
            avlastning_with_ageing',folder_root);
        project_name    = sprintf('run_%d-%s-%s_%s-%s-%s_%sLx-%d_Ly-%
            d_Lz-%d_%sFN-%d_mu_s-%s_mu_d-%s_%s%s_theta-%s_%s%sN-%d%s'
            ,...
                           year, month, day, hour, minute, second,...
                           str_run_time,Lx,Ly,Lz,...
                           str_v,total_normal_force,
                              str_static_friction_coefficient,
                              str_dynamic_friction_coefficient,...
                           str_sfsss,str_beta,str_theta,str_eta,
                              str_gamma,number_of_blocks,str_dt);
        folder_project  = sprintf('%s/%s',folder_driven, project_name)
            ;
        alternative_short = 'ALWA';
end


folder_code    = sprintf('%s/code',folder_project);
folder_data    = sprintf('%s/data',folder_project);
folder_info    = sprintf('%s/info',folder_project);
folder_log     = sprintf('%s/log',folder_project);
folder_analyze = sprintf('%s/analyze',folder_project);
folder_figure  = sprintf('%s/figure',folder_project);
fprintf(datestr(now,'dd mmm yyyy, HH:MM:SS\n'));
%-----------------------------------------------------------------------------

%   Creates a number of output files with the correct name and date
%-----------------------------------------------------------------------------


% Creates the project folder
[status,message,messageid] = mkdir(folder_project);
fprintf('Created directory %s\n',folder_project);
if status == 0
```

```
    fprintf('Something went wrong when creating project folder');
    break
end

% Creates the folder "code"
[status,message,messageid] = mkdir(folder_code);
fprintf('Created directory %s\n',folder_code);
if status == 0
    fprintf('Something went wrong when creating the folder Code');
    break
end

% Creates the folder "data"
[status,message,messageid] = mkdir(folder_data);
fprintf('Created directory %s\n',folder_data);
if status == 0
    fprintf('Something went wrong when creating the folder Data');
    break
end

% Unsync "data" from dropbox
cmd = sprintf('dropbox exclude add %s',folder_data);
system(cmd)
% pause(5)

% Creates the folder "data"
[status,message,messageid] = mkdir(folder_data);
fprintf('Created directory %s\n',folder_data);
if status == 0
    fprintf('Something went wrong when creating the folder Data');
    break
end

% Creates the folder "data/mat"
folder_data_mat = sprintf('%s/mat',folder_data);
[status,message,messageid] = mkdir(folder_data_mat);
fprintf('Created directory %s\n',folder_data_mat);
if status == 0
    fprintf('Something went wrong when creating the folder Data/mat');
    break
end

% Creates the folder "info"
[status,message,messageid] = mkdir(folder_info);
fprintf('Created directory %s\n',folder_info);
if status == 0
    fprintf('Something went wrong when creating the folder Data');
    break
end

% Creates the folder "type"
```

```
[status,message,messageid] = mkdir(folder_driven);
fprintf('Created directory %s\n',folder_driven);
if status == 0
    fprintf('Something went wrong when creating the folder Data');
    break
end

% Creates the folder "log"
[status,message,messageid] = mkdir(folder_log);
fprintf('Created directory %s\n',folder_log);
if status == 0
    fprintf('Something went wrong when creating the folder Log');
    break
end

% Unsync "log" from dropbox
cmd = sprintf('dropbox exclude add %s',folder_log);
system(cmd)
% pause(5)

% Creates the folder "log"
[status,message,messageid] = mkdir(folder_log);
fprintf('Created directory %s\n',folder_log);
if status == 0
    fprintf('Something went wrong when creating the folder Log');
    break
end

% Creates the folder "analyze"
[status,message,messageid] = mkdir(folder_analyze);
fprintf('Created directory %s\n',folder_analyze);
if status == 0
    fprintf('Something went wrong when creating the folder Analyze');
    break
end

% Creates the folder "figure"
[status,message,messageid] = mkdir(folder_figure);
fprintf('Created directory %s\n',folder_figure);
if status == 0
    fprintf('Something went wrong when creating the folder Figure');
    break
end

% Creates the folder "figure/All_forces"
folder_figure_all_forces = sprintf('%s/All_forces',folder_figure);
[status,message,messageid] = mkdir(folder_figure_all_forces);
fprintf('Created directory %s\n',folder_figure_all_forces);
if status == 0
    fprintf('Something went wrong when creating the folder Figure/
       All_forces');
    break
end

% Creates the folder "figure/All_forces"
folder_figure_all_forces_fig = sprintf('%s/All_forces/fig',
   folder_figure);
[status,message,messageid] = mkdir(folder_figure_all_forces_fig);
fprintf('Created directory %s\n',folder_figure_all_forces_fig);
if status == 0
    fprintf('Something went wrong when creating the folder Figure/
       All_forces/fig');
    break
end

% Creates the folder "figure/Damping_force"
folder_figure_damping_force = sprintf('%s/Damping_force',folder_figure
   );
[status,message,messageid] = mkdir(folder_figure_damping_force);
fprintf('Created directory %s\n',folder_figure_damping_force);
if status == 0
    fprintf('Something went wrong when creating the folder Figure/
       Damping_force');
    break
end

% Creates the folder "figure/Damping_force/fig"
folder_figure_damping_force_fig = sprintf('%s/Damping_force/fig',
   folder_figure);
[status,message,messageid] = mkdir(folder_figure_damping_force_fig);
fprintf('Created directory %s\n',folder_figure_damping_force_fig);
if status == 0
    fprintf('Something went wrong when creating the folder Figure/
       Damping_force/fig');
    break
end

% Creates the folder "figure/Friction_force"
folder_figure_friction_force = sprintf('%s/Friction_force',
   folder_figure);
[status,message,messageid] = mkdir(folder_figure_friction_force);
fprintf('Created directory %s\n',folder_figure_friction_force);
if status == 0
    fprintf('Something went wrong when creating the folder Figure/
       Friction_force');
    break
end

% Creates the folder "figure/Friction_force/fig"
folder_figure_friction_force_fig = sprintf('%s/Friction_force/fig',
   folder_figure);
[status,message,messageid] = mkdir(folder_figure_friction_force_fig);
```

```
fprintf('Created directory %s\n',folder_figure_friction_force_fig);
if status == 0
    fprintf('Something went wrong when creating the folder Figure/
       Friction_force/fig');
    break
end

% Creates the folder "figure/Fx"
folder_figure_fx = sprintf('%s/Fx',folder_figure);
[status,message,messageid] = mkdir(folder_figure_fx);
fprintf('Created directory %s\n',folder_figure_fx);
if status == 0
    fprintf('Something went wrong when creating the folder Figure/Fx')
       ;
    break
end

% Creates the folder "figure/Fx/fig"
folder_figure_fx_fig = sprintf('%s/Fx/fig',folder_figure);
[status,message,messageid] = mkdir(folder_figure_fx_fig);
fprintf('Created directory %s\n',folder_figure_fx_fig);
if status == 0
    fprintf('Something went wrong when creating the folder Figure/Fx/
       fig');
    break
end

% Creates the folder "figure/Lp_L-FT_FN"
folder_figure_Lp_L_FT_FN = sprintf('%s/Lp_L-FT_FN',folder_figure);
[status,message,messageid] = mkdir(folder_figure_Lp_L_FT_FN);
fprintf('Created directory %s\n',folder_figure_Lp_L_FT_FN);
if status == 0
    fprintf('Something went wrong when creating the folder Figure/Fx')
       ;
    break
end

% Creates the folder "figure/Lp_L-FT_FN/fig"
folder_figure_Lp_L_FT_FN_fig = sprintf('%s/Lp_L-FT_FN/fig',
   folder_figure);
[status,message,messageid] = mkdir(folder_figure_Lp_L_FT_FN_fig);
fprintf('Created directory %s\n',folder_figure_Lp_L_FT_FN_fig);
if status == 0
    fprintf('Something went wrong when creating the folder Figure/Fx/
       fig');
    break
end

% Creates the folder "figure/Position"
folder_figure_position = sprintf('%s/Position',folder_figure);
[status,message,messageid] = mkdir(folder_figure_position);
```

```
fprintf('Created directory %s\n',folder_figure_position);
if status == 0
    fprintf('Something went wrong when creating the folder Figure/
       Position');
    break
end

% Creates the folder "figure/Position/fig"
folder_figure_position_fig = sprintf('%s/Position/fig',folder_figure);
[status,message,messageid] = mkdir(folder_figure_position_fig);
fprintf('Created directory %s\n',folder_figure_position_fig);
if status == 0
    fprintf('Something went wrong when creating the folder Figure/
       Position/fig');
    break
end

% Creates the folder "figure/Shear_force"
folder_figure_shear_force = sprintf('%s/Shear_force',folder_figure);
[status,message,messageid] = mkdir(folder_figure_shear_force);
fprintf('Created directory %s\n',folder_figure_shear_force);
if status == 0
    fprintf('Something went wrong when creating the folder Figure/
       Shear_force');
    break
end

% Creates the folder "figure/Shear_force/fig"
folder_figure_shear_force_fig = sprintf('%s/Shear_force/fig',
   folder_figure);
[status,message,messageid] = mkdir(folder_figure_shear_force_fig);
fprintf('Created directory %s\n',folder_figure_shear_force_fig);
if status == 0
    fprintf('Something went wrong when creating the folder Figure/
       Shear_force/fig');
    old
end

% Creates the folder "figure/Velocity"
folder_figure_velocity = sprintf('%s/Velocity',folder_figure);
[status,message,messageid] = mkdir(folder_figure_velocity);
fprintf('Created directory %s\n',folder_figure_velocity);
if status == 0
    fprintf('Something went wrong when creating the folder Figure/
       Velocity');
    break
end

% Creates the folder "figure/Velocity/fig"
folder_figure_velocity_fig = sprintf('%s/Velocity/fig',folder_figure);
[status,message,messageid] = mkdir(folder_figure_velocity_fig);
```

```
fprintf('Created directory %s\n',folder_figure_velocity_fig);
if status == 0
    fprintf('Something went wrong when creating the folder Figure/
       Velocity/fig');
    old
end

% Creates the folder "figure/Normal_force"
folder_figure_normal_force = sprintf('%s/Normal_force',folder_figure);
[status,message,messageid] = mkdir(folder_figure_normal_force);
fprintf('Created directory %s\n',folder_figure_normal_force);
if status == 0
    fprintf('Something went wrong when creating the folder Figure/
       Normal_force');
    break
end

% Creates the folder "figure/Normal_force/fig"
folder_figure_normal_force_fig = sprintf('%s/Normal_force/fig',
   folder_figure);
[status,message,messageid] = mkdir(folder_figure_normal_force_fig);
fprintf('Created directory %s\n',folder_figure_normal_force_fig);
if status == 0
    fprintf('Something went wrong when creating the folder Figure/
       Normal_force/fig');
    old
end

switch alternative
    case 'side_driven'
        cd (src_driven);
        mex side_driven_mex.c
        copyfile('*.m',folder_code);
        copyfile('*.mexa64',folder_code);
        copyfile('*.c',folder_code);
        cd (src_analyze)
        copyfile('*.m',folder_analyze);
        cd (src_driven);
    case 'top_driven'
        cd (src_driven);
        mex top_driven_mex.c
        copyfile('*.m',folder_code);
        copyfile('*.mexa64',folder_code);
        copyfile('*.c',folder_code);
        cd (src_analyze)
        copyfile('*.m',folder_analyze);
        cd (src_driven);
    case 'left_right_driven'
        cd (src_driven);
        mex left_right_driven_mex.c
        copyfile('*.m',folder_code);
        copyfile('*.mexa64',folder_code);
        copyfile('*.c',folder_code);
        cd (src_analyze)
        copyfile('*.m',folder_analyze);
        cd (src_driven);
    case 'avlastning'
        cd (src_driven);
        mex avlastning_mex.c
        copyfile('*.mat',folder_code);
        copyfile('*.m',folder_code);
        copyfile('*.mexa64',folder_code);
        copyfile('*.c',folder_code);
        cd (src_analyze)
        copyfile('*.m',folder_analyze);
        cd (src_driven);
    case 'avlastning_desired_driving_force'
        cd (src_driven);
        mex avlastning_desired_driving_force_mex.c
        copyfile('*.m',folder_code);
        copyfile('*.mexa64',folder_code);
        copyfile('*.c',folder_code);
        cd (src_analyze)
        copyfile('*.m',folder_analyze);
        cd (src_driven);
    case 'avlastning_top_driven'
        cd (src_driven);
        mex avlastning_top_driven_mex.c
        copyfile('*.m',folder_code);
        copyfile('*.mexa64',folder_code);
        copyfile('*.c',folder_code);
        cd (src_analyze)
        copyfile('*.m',folder_analyze);
        cd (src_driven);
    case 'avlastning_top_driven_with_poisson'
        cd (src_driven);
        mex avlastning_top_driven_with_poisson_mex.c
        copyfile('*.m',folder_code);
        copyfile('*.mexa64',folder_code);
        copyfile('*.c',folder_code);
        cd (src_analyze)
        copyfile('*.m',folder_analyze);
        cd (src_driven);
    case 'avlastning_with_poisson'
        cd (src_driven);
        mex avlastning_with_poisson_mex.c
        copyfile('*.m',folder_code);
        copyfile('*.mexa64',folder_code);
        copyfile('*.c',folder_code);
        cd (src_analyze)
        copyfile('*.m',folder_analyze);
        cd (src_driven);
```

```
     case 'avlastning_with_ageing'
         cd (src_driven);
         mex avlastning_with_ageing_mex.c
         copyfile('*.m',folder_code);
         copyfile('*.mexa64',folder_code);
         copyfile('*.c',folder_code);
         cd (src_analyze)
         copyfile('*.m',folder_analyze);
         cd (src_driven);
end


% Create the output files
str_position                      = sprintf('%s/position.bin',
    folder_data);
str_velocity                      = sprintf('%s/velocity.bin',
    folder_data);
str_total_force                   = sprintf('%s/total_force.bin',
    folder_data);
str_driving_force                 = sprintf('%s/driving_force.bin',
    folder_data);
str_friction_force                = sprintf('%s/friction_force.bin',
    folder_data);
str_block_spring_force            = sprintf('%s/block_spring_force.bin
    ',folder_data);
str_damping_force                 = sprintf('%s/damping_force.bin',
    folder_data);
str_static_friction               = sprintf('%s/static_friction.bin',
    folder_data);
str_sliding                       = sprintf('%s/sliding.bin',
    folder_data);
str_length_static_friction_spring = sprintf('%s/
    length_static_friction_spring.bin',folder_data);
str_log                           = sprintf('%s/log.txt',folder_log);
str_initial                       = sprintf('%s/initial_conditions.txt
    ',folder_info);
str_folders                       = sprintf('%s/folders.txt',
    folder_info);
str_files                         = sprintf('%s/files.txt',folder_info
    );
str_plot_options                  = sprintf('%s/plot_options.txt',
    folder_info);
str_time                          = sprintf('%s/time.bin',folder_data)
    ;
str_normal_force                  = sprintf('%s/normal_force.bin',
    folder_data);


file_initial                       = fopen(str_initial,'w');
file_folders                       = fopen(str_folders,'w');
file_files                         = fopen(str_files,'w');
```

```
file_plot_options                  = fopen(str_plot_options,'w');
file_position                      = fopen(str_position,'w');
file_velocity                      = fopen(str_velocity,'w');
file_total_force                   = fopen(str_total_force,'w');
file_driving_force                 = fopen(str_driving_force,'w');
file_friction_force                = fopen(str_friction_force,'w');
file_spring_force                  = fopen(str_block_spring_force,'w')
   ;
file_damping_force                 = fopen(str_damping_force,'w');
file_static_friction               = fopen(str_static_friction,'w');
file_sliding                       = fopen(str_sliding,'w');
file_length_static_friction_spring = fopen(
   str_length_static_friction_spring,'w');
file_log                           = fopen(str_log,'w');
file_time                          = fopen(str_time,'w');
file_normal_force                  = fopen(str_normal_force,'w');

%---------------------------------------------------------------------------
%   Output initial conditions to file
%---------------------------------------------------------------------------


fprintf(file_initial,'run_time  = %.5f\n', run_time);
fprintf(file_initial,'dt = %.10f\n', dt);
fprintf(file_initial,'number_of_blocks  = %d\n', number_of_blocks);
fprintf(file_initial,'Lx = %d\n', Lx);
fprintf(file_initial,'Ly = %d\n', Ly);
fprintf(file_initial,'Lz = %d\n', Lz);
fprintf(file_initial,'total_system_mass  = %.5f\n', total_system_mass)
   ;
fprintf(file_initial,'youngs_modulus  = %.5f\n', youngs_modulus);
fprintf(file_initial,'relative_viscous_damping = %.10f\n',
   relative_viscous_damping);
fprintf(file_initial,'driving_spring_stiffness = %.5f\n',
   driving_spring_stiffness);
fprintf(file_initial,'total_normal_force = %.5f\n', total_normal_force
   );
fprintf(file_initial,'driving_velocity = %.5f\n', driving_velocity);
fprintf(file_initial,'static_friction_coefficient = %.5f\n',
   static_friction_coefficient);
fprintf(file_initial,'delta_static_friction_coefficient = %.5f\n',
   delta_static_friction_coefficient);
fprintf(file_initial,'dynamic_friction_coefficient = %.5f\n',
   dynamic_friction_coefficient);
fprintf(file_initial,'delta_dynamic_friction_coefficient = %.5f\n',
   delta_dynamic_friction_coefficient);
fprintf(file_initial,'theta = %.5f\n', theta);
fprintf(file_initial,'static_friction_spring_stiffness_scaling = %.5f\
   n', static_friction_spring_stiffness_scaling);
fprintf(file_initial,'beta = %.5f\n', beta);
fprintf(file_initial,'gamma = %.5f\n', gamma);
fprintf(file_initial,'driving_first_node = %.5f\n', driving_first_node
   );
fprintf(file_initial,'driving_last_node = %.5f', driving_last_node);
fprintf(file_initial,'multiplier = %.5f', multiplier);

fprintf(file_plot_options,'print_node = %d\n', print_node);
fprintf(file_plot_options,'print_freq = %d\n', print_freq);
fprintf(file_plot_options,'print_counter_limit = %d\n',
   print_counter_limit);
fprintf(file_plot_options,'print_counter_2_limit = %d\n',
   print_counter_2_limit);

fprintf(file_folders,'%s\n', folder_root);
fprintf(file_folders,'%s\n', src_driven);
fprintf(file_folders,'%s\n', src_analyze);
fprintf(file_folders,'%s\n', folder_driven);
fprintf(file_folders,'%s\n', folder_project);
fprintf(file_folders,'%s\n', folder_code);
fprintf(file_folders,'%s\n', folder_data);
fprintf(file_folders,'%s\n', folder_log);
fprintf(file_folders,'%s\n', folder_analyze);
fprintf(file_folders,'%s\n', folder_figure);
fprintf(file_folders,'%s\n', folder_info);
fprintf(file_folders,'%s\n', project_name);
fprintf(file_folders,'%s\n', alternative_short);

fprintf(file_files,'%s\n', str_position);
fprintf(file_files,'%s\n', str_velocity);
fprintf(file_files,'%s\n', str_total_force);
fprintf(file_files,'%s\n', str_driving_force);
fprintf(file_files,'%s\n', str_friction_force);
fprintf(file_files,'%s\n', str_block_spring_force);
fprintf(file_files,'%s\n', str_damping_force);
fprintf(file_files,'%s\n', str_static_friction);
fprintf(file_files,'%s\n', str_sliding);
fprintf(file_files,'%s\n', str_length_static_friction_spring);
fprintf(file_files,'%s\n', str_log);
fprintf(file_files,'%s\n', str_initial);
fprintf(file_files,'%s\n', str_folders);
fprintf(file_files,'%s\n', str_files);
fprintf(file_files,'%s\n', str_plot_options);
fprintf(file_files,'%s\n', str_time);
fprintf(file_files,'%s\n', str_normal_force);
fclose all;

%----------------------------------------------------------------------------

%   Run scripts
%----------------------------------------------------------------------------
```

```
switch alternative
    case 'side_driven'
        cd (folder_code)
        run side_driven
    case 'top_driven'
        cd (folder_code)
        run top_driven
    case 'left_right_driven'
        cd (folder_code)
        run left_right_driven
    case 'avlastning'
        cd (folder_code)
        run avlastning
    case 'avlastning_desired_driving_force'
        cd (folder_code)
        run avlastning_desired_driving_force
    case 'avlastning_top_driven'
        cd (folder_code)
        run avlastning_top_driven
    case 'avlastning_top_driven_with_poisson'
        cd (folder_code)
        run avlastning_top_driven_with_poisson
    case 'avlastning_with_poisson'
        cd (folder_code)
        run avlastning_with_poisson
    case 'avlastning_with_ageing'
        cd (folder_code)
        run avlastning_with_ageing
end
```

B.2.2 The time-loop code

The time-loop code is the second stage of the process of doing a simulation. There are one Matlab for each simulation we would like to run. We'll only show one of the because they are quite similar

```
clc
cd ..
cd info
import_initial = importdata('initial_conditions.txt', ' ' ,0);
import_folders = importdata('folders.txt');
import_files   = importdata('files.txt');
import_plot    = importdata('plot_options.txt');
cd ..
cd code
%--------------------------------------------------------------------------------

% INITIAL CONDITIONS
%--------------------------------------------------------------------------------

run_time                                   = import_initial.data(1);
dt                                         = import_initial.data(2);
number_of_blocks                           = import_initial.data(3);
Lx                                         = import_initial.data(4);
Ly                                         = import_initial.data(5);
Lz                                         = import_initial.data(6);
total_system_mass                          = import_initial.data(7);
youngs_modulus                             = import_initial.data(8);
relative_viscous_damping_factor            = import_initial.data(9);
driving_spring_stiffness                   = import_initial.data(10);
total_normal_force                         = import_initial.data(11);
driving_velocity                           = import_initial.data(12);
static_friction_coefficient                = import_initial.data(13);
delta_static_friction_coefficient          = import_initial.data(14);
dynamic_friction_coefficient               = import_initial.data(15);
delta_dynamic_friction_coefficient         = import_initial.data(16);
theta                                      = import_initial.data(17);
static_friction_spring_stiffness_scaling = import_initial.data(18);
beta                                       = import_initial.data(19);

print_node                        = import_plot.data(1);
print_freq                        = import_plot.data(2);
print_counter_limit               = import_plot.data(3);
print_counter_2_limit             = import_plot.data(4);

folder_root                       = char(import_folders(1));
src_top_driven                    = char(import_folders(2));
src_analyze                       = char(import_folders(3));
folder_type                       = char(import_folders(4));
folder_project                    = char(import_folders(5));
folder_code                       = char(import_folders(6));
folder_data                       = char(import_folders(7));
```

```
folder_log                          = char(import_folders(8));
folder_analyze                      = char(import_folders(9));
folder_figure                       = char(import_folders(10));
folder_info                         = char(import_folders(11));

str_position                          = char(import_files(1));
str_velocity                          = char(import_files(2));
str_total_force                       = char(import_files(3));
str_driving_force                     = char(import_files(4));
str_friction_force                    = char(import_files(5));
str_block_spring_force                = char(import_files(6));
str_damping_force                     = char(import_files(7));
str_static_friction                   = char(import_files(8));
str_sliding                           = char(import_files(9));
str_length_static_friction_spring     = char(import_files(10));
str_log                               = char(import_files(11));
str_initial                           = char(import_files(12));
str_folders                           = char(import_files(13));
str_files                             = char(import_files(14));
str_plot_options                      = char(import_files(15));
str_time                              = char(import_files(16));

file_position                          = fopen(str_position,'w');
file_velocity                          = fopen(str_velocity,'w');
file_total_force                       = fopen(str_total_force,'w');
file_driving_force                     = fopen(str_driving_force,'w');
file_friction_force                    = fopen(str_friction_force,'w');
file_block_spring_force                = fopen(str_block_spring_force,'w')
    ;
file_damping_force                     = fopen(str_damping_force,'w');
file_static_friction                   = fopen(str_static_friction,'w');
file_sliding                           = fopen(str_sliding,'w');
file_length_static_friction_spring     = fopen(
    str_length_static_friction_spring,'w');
file_time                              = fopen(str_time,'w');
diary(str_log);

%-------------------------------------------------------------------------

% BLOCK GEOMETRY
%-------------------------------------------------------------------------

area_of_block             = Ly*Lz;
last_block                = number_of_blocks;
first_block               = 1;

%-------------------------------------------------------------------------

% FORCES
%-------------------------------------------------------------------------
```

```
driving_force            = zeros(1,number_of_blocks);
shear_force              = zeros(1,number_of_blocks);
block_spring_force       = zeros(1,number_of_blocks);
damping_force            = zeros(1,number_of_blocks);

%------------------------------------------------------------------------

% SLIDER SPRING
%------------------------------------------------------------------------

block_spring_stiffness   = (youngs_modulus*(number_of_blocks - 1)*
    area_of_block)/Lx; %       [kN/mm]

normal_force_per_block = zeros(1,number_of_blocks);
for i = 1:number_of_blocks
     normal_force_per_block(i) = (total_normal_force/number_of_blocks)
         *(1 - ((2*i - number_of_blocks - 1)/(number_of_blocks - 1))*
         theta);
end

%------------------------------------------------------------------------

% SI CONVERSION
%------------------------------------------------------------------------

block_spring_stiffness    = block_spring_stiffness*1000;
                                                            %   [kN/
    mm]-->[N/mm]
driving_spring_stiffness  = driving_spring_stiffness*1000;
                                                          %   [MN/m]
    -->[N/mm]
mass_per_block            = total_system_mass/number_of_blocks;
                                                     %              [
    kg]
relative_viscous_damping  = sqrt(1000)*relative_viscous_damping_factor*
    sqrt(block_spring_stiffness*mass_per_block); %            [kg/s]

%------------------------------------------------------------------------

% FRICTION FORCE
%------------------------------------------------------------------------

static_friction_spring_stiffness   = block_spring_stiffness*
    static_friction_spring_stiffness_scaling;
static_friction_force              = (static_friction_coefficient +
    delta_static_friction_coefficient)*normal_force_per_block;
dynamic_friction_force             = (dynamic_friction_coefficient +
    delta_dynamic_friction_coefficient)*normal_force_per_block;
is_sliding                         = zeros(1,number_of_blocks);
old_length_static_friction_spring  = zeros(1,number_of_blocks);
new_length_static_friction_spring  = zeros(1,number_of_blocks);
```

```
friction_force                          = zeros(1,number_of_blocks);

%---------------------------------------------------------------------------------------------

% TIME SETUP
%---------------------------------------------------------------------------------------------

dt_divided_by_mass = double(dt/mass_per_block);
time_steps         = double(run_time/dt);
time_step_start    = 1;

%---------------------------------------------------------------------------------------------

% POSITION/VELOCITY/ACCELERATION
%---------------------------------------------------------------------------------------------

initial_block_positions = linspace(0,Lx,number_of_blocks);
initial_block_distances = initial_block_positions(2) -
    initial_block_positions(1);
initial_velocity        = zeros(1,number_of_blocks);
initial_acceleration    = zeros(1,number_of_blocks);
old_position            = initial_block_positions;
old_velocity            = initial_velocity;
old_acceleration        = initial_acceleration;
new_position            = zeros(1,number_of_blocks);
new_velocity            = zeros(1,number_of_blocks);
new_acceleration        = zeros(1,number_of_blocks);
sliding                 = zeros(1,number_of_blocks);
save_data               = true;

STATIC_FRICTION         = ones(1,number_of_blocks);
print_counter           = 0;
print_counter_2         = 0;
cpu_time                = cputime;

%---------------------------------------------------------------------------------------------

% INITIAL SHEAR FORCE PROFILE
%---------------------------------------------------------------------------------------------

if false
    % Set the initial shear force profile
    initial_shear_force = zeros(number_of_blocks,1);
    for j = 1:number_of_blocks
        initial_shear_force(j) = beta*normal_force_per_block(j)*((2*(
            old_position(j)-Lx/2))/Lx);
    end

    % Find the new position of the blocks
    new_change_in_position    = zeros(1,number_of_blocks);
    new_change_in_position(1) = 0;
    new_change_in_position(2) = new_change_in_position(1) +
       initial_shear_force(1)/block_spring_stiffness;
    for i = 3:number_of_blocks
        new_change_in_position(i) = 2*new_change_in_position(i-1) -
           new_change_in_position(i-2) + initial_shear_force(i-1)/
           block_spring_stiffness;
    end
    new_position = old_position + new_change_in_position;

    %Check if the block spring force is corect
    block_spring_force(first_block) = block_spring_stiffness*((
       new_position(first_block + 1) - new_position(first_block)) -
       initial_block_distances);
    block_spring_force(last_block) = block_spring_stiffness*(
       initial_block_distances - (new_position(last_block) -
       new_position(last_block-1)));
    for i = 2:number_of_blocks-1
        block_spring_force(i) = block_spring_stiffness*(new_position(i
           +1) - 2*new_position(i) + new_position(i-1));
    end

    % Find the new length of the static friction spring
    for i = first_block:1:last_block
        new_length_static_friction_spring(i) = block_spring_force(i)/
           static_friction_spring_stiffness;
        friction_force(i)                         = -
           static_friction_spring_stiffness*
           new_length_static_friction_spring(i);
    end
    old_position = new_position;
    old_length_static_friction_spring =
       new_length_static_friction_spring;
end

t = 0;

fwrite(file_position,new_position,'float64');
fwrite(file_velocity,new_velocity,'float64');
fwrite(file_total_force,shear_force,'float64');
fwrite(file_driving_force,driving_force,'float64');
fwrite(file_friction_force,friction_force,'float64');
fwrite(file_block_spring_force,block_spring_force,'float64');
fwrite(file_damping_force,damping_force,'float64');
fwrite(file_static_friction,STATIC_FRICTION,'float64');
fwrite(file_sliding,sliding,'float64');
fwrite(file_length_static_friction_spring,
   new_length_static_friction_spring,'float64');
fwrite(file_time,t,'float64');

t                = 0;
t_in             = t;
```

```
t_in_end       = t + print_freq;
sum_sliding_in = 0;
error(1)       = 0;
tic
while (t <= time_steps && error(1) == 0)
    [new_position, new_velocity, shear_force, friction_force,
       block_spring_force, damping_force,...
    STATIC_FRICTION, sliding, new_length_static_friction_spring,
       driving_force, error(1), t_out, sum_sliding_out]...
                                        = side_driven_mex(...
                                          old_position, old_velocity,
                                             old_length_static_friction_spring
                                             ,...
                                          static_friction_force,
                                             dynamic_friction_force,
                                             STATIC_FRICTION,...
                                          relative_viscous_damping,
                                             block_spring_stiffness,
                                             static_friction_spring_stiffness
                                             ,...
                                          dt_divided_by_mass,
                                             initial_block_distances,
                                             driving_spring_stiffness,
                                              driving_velocity,
                                             number_of_blocks,...
                                          t_in_end, t_in, dt,
                                             sum_sliding_in);
    t              = t_out;
    t_in           = t_out;
    sum_sliding_in = sum_sliding_out;
    %--------------------------------------------------------------------

    %   Write the data binary to file
    %--------------------------------------------------------------------

    fwrite(file_position,new_position,'float64');
    fwrite(file_velocity,new_velocity,'float64');
    fwrite(file_total_force,shear_force,'float64');
    fwrite(file_driving_force,driving_force,'float64');
    fwrite(file_friction_force,friction_force,'float64');
    fwrite(file_block_spring_force,block_spring_force,'float64');
    fwrite(file_damping_force,damping_force,'float64');
    fwrite(file_static_friction,STATIC_FRICTION,'float64');
    fwrite(file_sliding,sliding,'float64');
    fwrite(file_length_static_friction_spring,
       new_length_static_friction_spring,'float64');
    fwrite(file_time,t,'float64');

    time_used = cputime-cpu_time;
    total_run_time = (time_used/t)*time_steps;
    h =  floor((total_run_time - time_used)/(60*60));
    min = floor(((total_run_time - time_used)/(60*60) - h)*60);
    sec = floor(((((total_run_time - time_used)/(60*60) - h)*60) - min
       )*60);

    if (t == t_in_end)
        t_in_end = t_in_end + print_freq;
        print_counter = print_counter + 1;
    else
        print_counter_2 = print_counter_2 + 1;
    end
    if ((print_counter == print_counter_limit || print_counter_2 ==
       print_counter_2_limit))
        if (print_counter_2 == print_counter_2_limit)
            print_counter_2 = 0;
        elseif (print_counter  == print_counter_limit)
            print_counter = 0;
        end
        distance_right_block = new_position(print_node+1) -
           new_position(print_node);
        distance_left_block  = new_position(print_node) - new_position
           (print_node-1);
        total_block_length   = new_position(last_block) - new_position
           (first_block);
        total_driving_force  = sum(driving_force);
        clc

        fprintf('Run time:                              %d [s]\n',
           run_time)
        fprintf('Time step:                             %d\n',t)
        fprintf('Real time:                             %.10f [s]\n',
           t*dt)
        fprintf('Sum sliding:                           %d\n',
           sum_sliding_in)
        fprintf('Block spring stiffness:                %.3f [N/mm]\n
           ',block_spring_stiffness)
        fprintf('Static friction spring stiffness:      %.3f [N/mm]\n
           ',static_friction_spring_stiffness)
        fprintf('Driving spring stiffness:              %.3f [N/mm]\n
           ',driving_spring_stiffness)
        fprintf('Relative viscous damping:              %.3f [kg/s]\n
           ',relative_viscous_damping)
        fprintf('Fx(t)/P:                               %.7f\n',
           total_driving_force/sum(normal_force_per_block))
        fprintf('Total driving force:                   %.3f [N]\n',
           total_driving_force)
        fprintf('Normal force (P):                      %.7f [N]\n',
           sum(normal_force_per_block))
        fprintf('
           ----------------------------------------------------------------------\
           n')
        fprintf('Driving force on node (%d):            %.3f [N]\n',
            first_block ,driving_force(first_block))
        fprintf('Damping force on node (%d):                %.10f [N]\n',
            first_block ,damping_force(first_block))
        fprintf('Spring force on node (%d):                 %.7f [N]\n',
            first_block ,block_spring_force(first_block))
        fprintf('Total foce on node (%d):                   %.7f [N]\n',
            first_block ,shear_force(first_block))
        fprintf('Friction force on node (%d):               %.7f [N]\n',
            first_block ,friction_force(first_block))
        fprintf('New velocity(%d):                          %.7f [mm/s]\n
            ',first_block ,new_velocity(first_block))
        fprintf('New position(%d):                          %.7f [mm]\n',
            first_block ,new_position(first_block))
        fprintf('Initial position(%d):                      %.7f [mm]\n',
            first_block ,initial_block_positions(first_block))
        fprintf('Length of friction spring (%d):            %.7f [mm]\n',
            first_block ,new_length_static_friction_spring(first_block))
        fprintf('Change in block position (%d):             %.7f [mm]\n',
            first_block ,new_position(first_block) -
            initial_block_positions(first_block))
        fprintf('
            ---------------------------------------------------------------------\
            n')
        fprintf('Damping force on node (%d):                %.10f [N]\n',
            print_node ,damping_force(print_node))
        fprintf('Spring force on node (%d):                 %.7f [N]\n',
            print_node ,block_spring_force(print_node))
        fprintf('Total foce on node (%d):                   %.7f [N]\n',
            print_node ,shear_force(print_node))
        fprintf('Friction force on node (%d):               %.7f [N]\n',
            print_node ,friction_force(print_node))
        fprintf('New velocity (%d):                         %.7f [mm/s]\n
            ',print_node ,new_velocity(print_node))
        fprintf('New position (%d):                         %.7f [mm]\n',
            print_node ,new_position(print_node))
        fprintf('Initial position (%d):                     %.7f [mm]\n',
            print_node ,initial_block_positions(print_node))
        fprintf('Length of friction spring (%d):            %.7f [mm]\n',
            print_node ,new_length_static_friction_spring(print_node))
        fprintf('Change in block position (%d):             %.7f [mm]\n',
            print_node ,new_position(print_node) -
            initial_block_positions(print_node))
        fprintf('
            ---------------------------------------------------------------------\
            n')
        fprintf('Driving force on node (%d):                %.3f [N]\n',
            last_block ,driving_force(last_block))
        fprintf('Damping force on node (%d):                %.10f [N]\n',
            last_block ,damping_force(last_block))
        fprintf('Spring force on node (%d):                 %.7f [N]\n',
            last_block ,block_spring_force(last_block))
        fprintf('Total foce on node (%d):                  %.7f [N]\n',
           last_block,shear_force(last_block))
        fprintf('Friction force on node (%d):              %.7f [N]\n',
           last_block,friction_force(last_block))
        fprintf('New velocity (%d):                        %.7f [mm/s]\n
           ',last_block,new_velocity(last_block))
        fprintf('New position (%d):                        %.7f [mm]\n',
           last_block,new_position(last_block))
        fprintf('Initial position (%d):                    %.7f [mm]\n',
           last_block,initial_block_positions(last_block))
        fprintf('Change in block position (%d):            %.7f [mm]\n',
           last_block,new_position(last_block) -
           initial_block_positions(last_block))
        fprintf('
           ----------------------------------------------------------------------\
           n')
        fprintf('Distance to left node:                    %.7f [mm]\n',
           distance_left_block)
        fprintf('Distance to right node:                   %.7f [mm]\n',
           distance_right_block)
        fprintf('Initial distance between nodes:           %.7f [mm]\n',
           initial_block_distances)
        fprintf('Total legnth of the block:                %.7f [mm]\n',
           total_block_length)
        fprintf('Change in block length:                   %.7f [mm]\n',
           total_block_length - initial_block_positions(last_block))
        fprintf('
           ----------------------------------------------------------------------\
           n')
        fprintf('Time used on %g out of %g iterations :  %d h %d m %d
           s\n',t,time_steps,h,min,sec);
        fprintf('
           ----------------------------------------------------------------------\
           n')
    end
    %------------------------------------------------------------------------

    % UPDATE OLD TO NEW
    %------------------------------------------------------------------------

    old_position = new_position;
    old_velocity = new_velocity;
    old_length_static_friction_spring =
       new_length_static_friction_spring;
end
toc
fclose all;
```

B.3 The C code for the side driven model

```
#include <math.h>
#include "mex.h"

/* Input Arguments */
#define IN_POSITION                              prhs[0]
#define IN_VELOCITY                              prhs[1]
#define IN_LENGTH_STATIC_FRICTION_SPRING         prhs[2]
#define IN_STATIC_FRICTION_FORCE                 prhs[3]
#define IN_DYNAMIC_FRICTION_FORCE                prhs[4]
#define IN_STATIC_FRICTION                       prhs[5]

#define IN_RELATIVE_VISCOUS_DAMPING              prhs[6]
#define IN_BLOCK_SPRING_STIFFNESS                prhs[7]
#define IN_STATIC_FRICTION_SPRING_STIFFNESS      prhs[8]
#define IN_DT_DIVIDED_BY_MASS                    prhs[9]
#define IN_INITIAL_BLOCK_DISTANCES               prhs[10]
#define IN_DRIVING_SPRING_STIFFNESS              prhs[11]
#define IN_DRIVING_VELOCITY                      prhs[12]
#define IN_NUMBER_OF_BLOCKS                      prhs[13]
#define IN_T_END                                 prhs[14]
#define IN_T                                     prhs[15]
#define IN_DT                                    prhs[16]
#define IN_SUM_SLIDING                           prhs[17]

/* Output Arguments */
#define OUT_POSITION                              plhs[0]
#define OUT_VELOCITY                              plhs[1]
#define OUT_SHEAR_FORCE                           plhs[2]
#define OUT_FRICTION_FORCE                        plhs[3]
#define OUT_BLOCK_SPRING_FORCE                    plhs[4]
#define OUT_DAMPING_FORCE                         plhs[5]
#define OUT_STATIC_FRICTION                       plhs[6]
#define OUT_SLIDING                               plhs[7]
#define OUT_LENGTH_STATIC_FRICTION_SPRING         plhs[8]
#define OUT_DRIVING_FORCE                         plhs[9]
#define OUT_ERROR                                 plhs[10]
#define OUT_T                                     plhs[11]
#define OUT_SUM_SLIDING                           plhs[12]

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray*
    prhs[]){
    /* Input */
    double *in_velocity, *in_position, *
        in_length_static_friction_spring;
    double *in_static_friction_force, *in_dynamic_friction_force, *
        in_static_friction;
    double in_relative_viscous_damping;
    double in_block_spring_stiffness,
        in_static_friction_spring_stiffness;
```

```
double dt_divided_by_mass, initial_block_distances;
double driving_spring_stiffness, driving_velocity,
   number_of_blocks;
double dt;
double t, t_end;
double in_sum_sliding;

/* Output */
double *out_position, *out_velocity;
double *out_shear_force, *out_friction_force, *
   out_block_spring_force, *out_damping_force;
double *out_static_friction, *out_sliding, *
   out_length_static_friction_spring;
double *out_driving_force;
double *out_error;
double *out_t;
double *out_sum_sliding;

/* In use inside c-script */
int i, j, k;
int first_block, last_block;
double end_loop;
double sum_sliding;
double sum_sliding_counter;
double* tmp_old_position;
double* tmp_old_velocity;
double* tmp_old_length_static_friction_spring;
double* tmp_new_position;
double* tmp_new_velocity;
double* tmp_new_length_static_friction_spring;
double* tmp_static_friction;
double* tmp_driving_force;
double* tmp_damping_force;
double* tmp_block_spring_force;
double* tmp_shear_force;
double* tmp_friction_force;
double* tmp_sliding;

/* Input */
in_velocity                          = mxGetPr(IN_VELOCITY);
in_position                          = mxGetPr(IN_POSITION);
in_length_static_friction_spring     = mxGetPr(
   IN_LENGTH_STATIC_FRICTION_SPRING);
in_static_friction_force             = mxGetPr(
   IN_STATIC_FRICTION_FORCE);
in_dynamic_friction_force            = mxGetPr(
   IN_DYNAMIC_FRICTION_FORCE);
in_static_friction                   = mxGetPr(IN_STATIC_FRICTION
   );
```

```
    in_relative_viscous_damping            = mxGetScalar(
        IN_RELATIVE_VISCOUS_DAMPING);
    in_block_spring_stiffness              = mxGetScalar(
        IN_BLOCK_SPRING_STIFFNESS);
    in_static_friction_spring_stiffness    = mxGetScalar(
        IN_STATIC_FRICTION_SPRING_STIFFNESS);
    dt_divided_by_mass                     = mxGetScalar(
        IN_DT_DIVIDED_BY_MASS);
    initial_block_distances                = mxGetScalar(
        IN_INITIAL_BLOCK_DISTANCES);
    driving_spring_stiffness               = mxGetScalar(
        IN_DRIVING_SPRING_STIFFNESS);
    driving_velocity                       = mxGetScalar(
        IN_DRIVING_VELOCITY);
    number_of_blocks                       = mxGetScalar(
        IN_NUMBER_OF_BLOCKS);
    t_end                                  = mxGetScalar(IN_T_END);
    t                                      = mxGetScalar(IN_T);
    dt                                     = mxGetScalar(IN_DT);
    in_sum_sliding                         = mxGetScalar(IN_SUM_SLIDING
        );

    /* Output */
    OUT_POSITION                           = mxCreateDoubleMatrix(1,
        number_of_blocks,mxREAL);
    OUT_VELOCITY                           = mxCreateDoubleMatrix(1,
        number_of_blocks,mxREAL);
    OUT_SHEAR_FORCE                        = mxCreateDoubleMatrix(1,
        number_of_blocks,mxREAL);
    OUT_FRICTION_FORCE                     = mxCreateDoubleMatrix(1,
        number_of_blocks,mxREAL);
    OUT_BLOCK_SPRING_FORCE                 = mxCreateDoubleMatrix(1,
        number_of_blocks,mxREAL);
    OUT_DAMPING_FORCE                      = mxCreateDoubleMatrix(1,
        number_of_blocks,mxREAL);
    OUT_STATIC_FRICTION                    = mxCreateDoubleMatrix(1,
        number_of_blocks,mxREAL);
    OUT_SLIDING                            = mxCreateDoubleMatrix(1,
        number_of_blocks,mxREAL);
    OUT_LENGTH_STATIC_FRICTION_SPRING      = mxCreateDoubleMatrix(1,
        number_of_blocks,mxREAL);
    OUT_DRIVING_FORCE                      = mxCreateDoubleMatrix(1,
        number_of_blocks,mxREAL);
    OUT_ERROR                              = mxCreateDoubleMatrix(1,1,
        mxREAL);
    OUT_T                                  = mxCreateDoubleMatrix(1,1,
        mxREAL);
    OUT_SUM_SLIDING                        = mxCreateDoubleMatrix(1,1,
        mxREAL);

    out_velocity                           = mxGetPr(OUT_VELOCITY);
```

```
out_position                        = mxGetPr(OUT_POSITION);
out_shear_force                     = mxGetPr(OUT_SHEAR_FORCE);
out_friction_force                  = mxGetPr(OUT_FRICTION_FORCE
    );
out_block_spring_force              = mxGetPr(
    OUT_BLOCK_SPRING_FORCE);
out_damping_force                   = mxGetPr(OUT_DAMPING_FORCE)
    ;
out_static_friction                 = mxGetPr(
    OUT_STATIC_FRICTION);
out_sliding                         = mxGetPr(OUT_SLIDING);
out_length_static_friction_spring   = mxGetPr(
    OUT_LENGTH_STATIC_FRICTION_SPRING);
out_driving_force                   = mxGetPr(OUT_DRIVING_FORCE)
    ;
out_error                           = mxGetPr(OUT_ERROR);
out_t                               = mxGetPr(OUT_T);
out_sum_sliding                     = mxGetPr(OUT_SUM_SLIDING);

/* In use inside c-script */
end_loop                            = 0;
out_error[0]                        = 0;
sum_sliding_counter                 = 0;
first_block                         = 0;
last_block                          = number_of_blocks-1;

tmp_old_velocity                    = (double *)malloc(
    number_of_blocks * sizeof(double));
tmp_old_position                    = (double *)malloc(
    number_of_blocks * sizeof(double));
tmp_new_velocity                    = (double *)malloc(
    number_of_blocks * sizeof(double));
tmp_new_position                    = (double *)malloc(
    number_of_blocks * sizeof(double));
tmp_old_length_static_friction_spring = (double *)malloc(
    number_of_blocks * sizeof(double));
tmp_new_length_static_friction_spring = (double *)malloc(
    number_of_blocks * sizeof(double));
tmp_static_friction                 = (double *)malloc(
    number_of_blocks * sizeof(double));
tmp_driving_force                   = (double *)malloc(
    number_of_blocks * sizeof(double));
tmp_damping_force                   = (double *)malloc(
    number_of_blocks * sizeof(double));
tmp_block_spring_force              = (double *)malloc(
    number_of_blocks * sizeof(double));
tmp_shear_force                     = (double *)malloc(
    number_of_blocks * sizeof(double));
tmp_friction_force                  = (double *)malloc(
    number_of_blocks * sizeof(double));
    tmp_sliding                                  = (double *)malloc(
        number_of_blocks * sizeof(double));

    for (j = 0; j < number_of_blocks; j++){
        tmp_old_velocity[j]                          = in_velocity[j];
        tmp_old_position[j]                          = in_position[j];
        tmp_old_length_static_friction_spring[j] =
            in_length_static_friction_spring[j];
        tmp_static_friction[j]                       = in_static_friction[
            j];
        tmp_driving_force[j]                         = 0;
    }

    while (t < t_end && end_loop == 0 && out_error[0] == 0){
        t              = t + 1;
        sum_sliding    = 0;
        /* Calculate the first block */
        tmp_driving_force[first_block]        = driving_spring_stiffness
            *(driving_velocity*(dt*t) - tmp_old_position[first_block]);
        tmp_damping_force[first_block]        =
            in_relative_viscous_damping*(tmp_old_velocity[first_block +
             1] - tmp_old_velocity[first_block]);
        tmp_block_spring_force[first_block] =
            in_block_spring_stiffness*((tmp_old_position[first_block +
            1] - tmp_old_position[first_block]) -
            initial_block_distances);
        tmp_shear_force[first_block]          = tmp_damping_force[
            first_block] + tmp_block_spring_force[first_block] +
            tmp_driving_force[first_block];
        if (tmp_static_friction[first_block] == 1){
            /* Static friction */
            tmp_friction_force[first_block] = -
                in_static_friction_spring_stiffness*
                tmp_old_length_static_friction_spring[first_block];
            tmp_new_velocity[first_block]     = tmp_old_velocity[
                first_block] + (tmp_shear_force[first_block] +
                tmp_friction_force[first_block])*dt_divided_by_mass;
            tmp_new_position[first_block]     = tmp_old_position[
                first_block] + tmp_new_velocity[first_block]*dt;
            tmp_sliding[first_block]                            = 0;
            tmp_static_friction[first_block]                    = 1;
            tmp_new_length_static_friction_spring[first_block] =
                tmp_old_length_static_friction_spring[first_block] +
                tmp_new_position[first_block] - tmp_old_position[
                first_block];
            if (fabs(tmp_shear_force[first_block]) >=
                in_static_friction_force[first_block]){
                /*tmp_sliding[first_block]          = 1;*/
                /*sum_sliding                      += 1;*/
                tmp_static_friction[first_block]   = 0;
            }
    }
    else if (tmp_static_friction[first_block] == 0){
        /* Dynamic friction */
        tmp_friction_force[first_block] = -
            in_dynamic_friction_force[first_block]*(
            tmp_old_velocity[first_block]/fabs(tmp_old_velocity[
            first_block]));
        tmp_new_velocity[first_block]    = tmp_old_velocity[
            first_block] + (tmp_shear_force[first_block] +
            tmp_friction_force[first_block])*dt_divided_by_mass;
        tmp_new_position[first_block]    = tmp_old_position[
            first_block] + tmp_new_velocity[first_block]*dt;
        tmp_sliding[first_block]                              = 1;
        sum_sliding                                          += 1;
        tmp_static_friction[first_block]                      = 0;
        tmp_new_length_static_friction_spring[first_block] = 0;
        if (tmp_new_velocity[first_block] <= 0){
            /*tmp_sliding[first_block]                            =
                0;*/
            /*sum_sliding                                        -=
                1;*/
            tmp_static_friction[first_block]                    =
                1;
            tmp_new_length_static_friction_spring[first_block] =
                tmp_shear_force[first_block]/
                in_static_friction_spring_stiffness;
        }
    }
    else{
        out_error[0] = 1;
        mexPrintf("ERROR at t = %d [s] for block %d static
            friction = %.1f\n",t,first_block,tmp_static_friction[
            first_block]);
    }
    /* Calculate the last block */
    tmp_block_spring_force[last_block] = in_block_spring_stiffness
        *(initial_block_distances - (tmp_old_position[last_block] -
         tmp_old_position[last_block-1]));
    tmp_damping_force[last_block]      =
        in_relative_viscous_damping*(tmp_old_velocity[last_block-1]
        - tmp_old_velocity[last_block]);
    tmp_shear_force[last_block]         = tmp_damping_force[
        last_block] + tmp_block_spring_force[last_block];
    if (tmp_static_friction[last_block] == 1){
        /* Static friction */
        tmp_friction_force[last_block] = -
            in_static_friction_spring_stiffness*
            tmp_old_length_static_friction_spring[last_block];
        tmp_new_velocity[last_block]    = tmp_old_velocity[
            last_block] + (tmp_shear_force[last_block] +
            tmp_friction_force[last_block])*dt_divided_by_mass;
```

```
                tmp_new_position[last_block]    = tmp_old_position[
                    last_block] + tmp_new_velocity[last_block]*dt;
                tmp_sliding[last_block]                           = 0;
                tmp_static_friction[last_block]                   = 1;
                tmp_new_length_static_friction_spring[last_block] =
                    tmp_old_length_static_friction_spring[last_block] +
                    tmp_new_position[last_block] - tmp_old_position[
                    last_block];
                if (fabs(tmp_shear_force[last_block]) >=
                    in_static_friction_force[last_block]){
                    /*tmp_sliding[last_block]          = 1;*/
                    /*sum_sliding                     += 1;*/
                    tmp_static_friction[last_block] = 0;
                }
            }
            else if (tmp_static_friction[last_block] == 0){
                /* Dynamic friction */
                tmp_friction_force[last_block] = -
                    in_dynamic_friction_force[last_block]*(tmp_old_velocity
                    [last_block]/fabs(tmp_old_velocity[last_block]));
                tmp_new_velocity[last_block]    = tmp_old_velocity[
                    last_block] + (tmp_shear_force[last_block] +
                    tmp_friction_force[last_block])*dt_divided_by_mass;
                tmp_new_position[last_block]    = tmp_old_position[
                    last_block] + tmp_new_velocity[last_block]*dt;
                tmp_sliding[last_block]                           = 1;
                sum_sliding                                      += 1;
                tmp_static_friction[last_block]                   = 0;
                tmp_new_length_static_friction_spring[last_block] = 0;
                if (tmp_new_velocity[last_block] <= 0){
                    /*tmp_sliding[last_block]                           =
                        0;*/
                    /*sum_sliding                                      -=
                        1;*/
                    tmp_static_friction[last_block]                   = 1;
                    tmp_new_length_static_friction_spring[last_block] =
                        tmp_shear_force[last_block]/
                        in_static_friction_spring_stiffness;
                }
            }
            else{
                out_error[0] = 1;
                mexPrintf("ERROR at t = %d [s] for block %d static
                    friction = %.1f\n",t,last_block,tmp_static_friction[
                    last_block]);
            }
            for (i = first_block + 1; i < last_block; i++){
                /* Calculate the midle blocks */
                tmp_damping_force[i]       = in_relative_viscous_damping*(
                    tmp_old_velocity[i+1] - 2*tmp_old_velocity[i] +
                    tmp_old_velocity[i-1]);
```

```
        tmp_block_spring_force[i] = in_block_spring_stiffness*(
            tmp_old_position[i+1] - 2*tmp_old_position[i] +
            tmp_old_position[i-1]);
        tmp_shear_force[i]        = tmp_damping_force[i] +
            tmp_block_spring_force[i];
        if (tmp_static_friction[i] == 1){
            /* Static friction */
            tmp_friction_force[i] = -
                in_static_friction_spring_stiffness*
                tmp_old_length_static_friction_spring[i];
            tmp_new_velocity[i]   = tmp_old_velocity[i] + (
                tmp_shear_force[i] + tmp_friction_force[i])*
                dt_divided_by_mass;
            tmp_new_position[i]   = tmp_old_position[i] +
                tmp_new_velocity[i]*dt;
            tmp_sliding[i]                           = 0;
            tmp_static_friction[i]                   = 1;
            tmp_new_length_static_friction_spring[i] =
                tmp_old_length_static_friction_spring[i] +
                tmp_new_position[i] - tmp_old_position[i];
            if (fabs(tmp_shear_force[i]) >=
                in_static_friction_force[i]){
                /*tmp_sliding[i]          = 1;*/
                /*sum_sliding            += 1;*/
                tmp_static_friction[i] = 0;
            }
        }
        else if (tmp_static_friction[i] == 0){
            /* Dynamic friction */
            tmp_friction_force[i] = -in_dynamic_friction_force[i
                ]*(tmp_old_velocity[i]/fabs(tmp_old_velocity[i]));
            tmp_new_velocity[i]   = tmp_old_velocity[i] + (
                tmp_shear_force[i] + tmp_friction_force[i])*
                dt_divided_by_mass;
            tmp_new_position[i]   = tmp_old_position[i] +
                tmp_new_velocity[i]*dt;
            tmp_sliding[i]                           = 1;
            sum_sliding                             += 1;
            tmp_static_friction[i]                   = 0;
            tmp_new_length_static_friction_spring[i] = 0;
            if (tmp_new_velocity[i] <= 0){
                /*tmp_sliding[i]                           = 0;*/
                /*sum_sliding                             -= 1;*/
                tmp_static_friction[i]                   = 1;
                tmp_new_length_static_friction_spring[i] =
                    tmp_shear_force[i]/
                    in_static_friction_spring_stiffness;
            }
        }
        else{
            out_error[0] = 1;
```

```
                    mexPrintf("ERROR at t = %d [s] for block %d static
                        friction = %.1f\n",t,i,tmp_static_friction[i]);
                }
            }
            for (j = 0; j < number_of_blocks;j++){
                tmp_old_position[j]                             =
                    tmp_new_position[j];
                tmp_old_velocity[j]                             =
                    tmp_new_velocity[j];
                tmp_old_length_static_friction_spring[j] =
                    tmp_new_length_static_friction_spring[j];
            }

            if (((sum_sliding > 0) && (sum_sliding != in_sum_sliding)) ||
               ((sum_sliding == 0) && (in_sum_sliding > 0))){
                end_loop          = 1;
            }
        }

        *out_sum_sliding = sum_sliding;
        *out_t             = t;
        for (k = 0; k < number_of_blocks;k++){
            out_static_friction[k]                  = tmp_static_friction[k];
            out_position[k]                         = tmp_new_position[k];
            out_velocity[k]                         = tmp_new_velocity[k];
            out_length_static_friction_spring[k] =
                tmp_new_length_static_friction_spring[k];
            out_driving_force[k]                    = tmp_driving_force[k];
            out_damping_force[k]                    = tmp_damping_force[k];
            out_block_spring_force[k]               = tmp_block_spring_force[
                k];
            out_shear_force[k]                      = tmp_shear_force[k];
            out_friction_force[k]                   = tmp_friction_force[k];
            out_sliding[k]                          = tmp_sliding[k];
        }

        free(tmp_old_velocity);
        free(tmp_old_position);
        free(tmp_old_length_static_friction_spring);
        free(tmp_new_velocity);
        free(tmp_new_position);
        free(tmp_new_length_static_friction_spring);
        free(tmp_static_friction);
        free(tmp_driving_force);
        free(tmp_damping_force);
        free(tmp_block_spring_force);
        free(tmp_shear_force);
        free(tmp_friction_force);
        free(tmp_sliding);
        return;
}
```

B.4 The C code for the top driven model

```
#include <math.h>
#include "mex.h"

/* Input Arguments */
#define IN_POSITION                             prhs[0]
#define IN_VELOCITY                             prhs[1]
#define IN_LENGTH_STATIC_FRICTION_SPRING        prhs[2]
#define IN_STATIC_FRICTION_FORCE                prhs[3]
#define IN_DYNAMIC_FRICTION_FORCE               prhs[4]
#define IN_STATIC_FRICTION                      prhs[5]
#define IN_INITIAL_POSITIONS                    prhs[6]

#define IN_RELATIVE_VISCOUS_DAMPING             prhs[7]
#define IN_BLOCK_SPRING_STIFFNESS               prhs[8]
#define IN_STATIC_FRICTION_SPRING_STIFFNESS     prhs[9]
#define IN_DT_DIVIDED_BY_MASS                   prhs[10]
#define IN_INITIAL_BLOCK_DISTANCES              prhs[11]
#define IN_DRIVING_SPRING_STIFFNESS             prhs[12]
#define IN_DRIVING_VELOCITY                     prhs[13]
#define IN_NUMBER_OF_BLOCKS                     prhs[14]
#define IN_T_END                                prhs[15]
#define IN_T                                    prhs[16]
#define IN_DT                                   prhs[17]
#define IN_SUM_SLIDING                          prhs[18]

/* Output Arguments */
#define OUT_NEW_POSITION                         plhs[0]
#define OUT_NEW_VELOCITY                         plhs[1]
#define OUT_SHEAR_FORCE                          plhs[2]
#define OUT_FRICTION_FORCE                       plhs[3]
#define OUT_BLOCK_SPRING_FORCE                   plhs[4]
#define OUT_DAMPING_FORCE                        plhs[5]
#define OUT_STATIC_FRICTION                      plhs[6]
#define OUT_SLIDING                              plhs[7]
#define OUT_NEW_LENGTH_STATIC_FRICTION_SPRING    plhs[8]
#define OUT_DRIVING_FORCE                        plhs[9]
#define OUT_ERROR                                plhs[10]
#define OUT_T                                    plhs[11]
#define OUT_SUM_SLIDING                          plhs[12]

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray*
    prhs[]){
    /* Input */
    double *in_velocity, *in_position, *
        in_length_static_friction_spring;
    double *in_static_friction_force, *in_dynamic_friction_force, *
        in_static_friction, *in_initial_block_positions;
    double in_relative_viscous_damping;
double in_block_spring_stiffness ,
   in_static_friction_spring_stiffness;
double dt_divided_by_mass , initial_block_distances;
double driving_spring_stiffness , driving_velocity ,
   number_of_blocks;
double dt;
double t , t_end;
double in_sum_sliding;

/* Output */
double *out_position , *out_velocity;
double *out_shear_force , *out_friction_force , *
   out_block_spring_force , *out_damping_force;
double *out_static_friction , *out_sliding , *
   out_length_static_friction_spring;
double *out_driving_force;
double *out_error;
double *out_t;
double *out_sum_sliding;

/* In use inside c-script */
int i , j , k;
int first_block , last_block;
double end_loop;
double sum_sliding;
double sum_sliding_counter;
double* tmp_old_position;
double* tmp_old_velocity;
double* tmp_old_length_static_friction_spring;
double* tmp_new_position;
double* tmp_new_velocity;
double* tmp_new_length_static_friction_spring;
double* tmp_static_friction;
double* tmp_driving_force;
double* tmp_damping_force;
double* tmp_block_spring_force;
double* tmp_shear_force;
double* tmp_friction_force;
double* tmp_sliding;

/* Input */
in_velocity                          = mxGetPr(IN_VELOCITY);
in_position                          = mxGetPr(IN_POSITION);
in_length_static_friction_spring     = mxGetPr(
   IN_LENGTH_STATIC_FRICTION_SPRING);
in_static_friction_force             = mxGetPr(
   IN_STATIC_FRICTION_FORCE);
in_dynamic_friction_force            = mxGetPr(
   IN_DYNAMIC_FRICTION_FORCE);
in_static_friction                   = mxGetPr(IN_STATIC_FRICTION
   );
```

```
    in_initial_block_positions              = mxGetPr(
        IN_INITIAL_POSITIONS);

    in_relative_viscous_damping             = mxGetScalar(
        IN_RELATIVE_VISCOUS_DAMPING);
    in_block_spring_stiffness               = mxGetScalar(
        IN_BLOCK_SPRING_STIFFNESS);
    in_static_friction_spring_stiffness     = mxGetScalar(
        IN_STATIC_FRICTION_SPRING_STIFFNESS);
    dt_divided_by_mass                      = mxGetScalar(
        IN_DT_DIVIDED_BY_MASS);
    initial_block_distances                 = mxGetScalar(
        IN_INITIAL_BLOCK_DISTANCES);
    driving_spring_stiffness                = mxGetScalar(
        IN_DRIVING_SPRING_STIFFNESS);
    driving_velocity                        = mxGetScalar(
        IN_DRIVING_VELOCITY);
    number_of_blocks                        = mxGetScalar(
        IN_NUMBER_OF_BLOCKS);
    t_end                                   = mxGetScalar(IN_T_END);
    t                                       = mxGetScalar(IN_T);
    dt                                      = mxGetScalar(IN_DT);
    in_sum_sliding                          = mxGetScalar(IN_SUM_SLIDING
        );

    /* Output */
    OUT_NEW_POSITION                        = mxCreateDoubleMatrix(1,
        number_of_blocks,mxREAL);
    OUT_NEW_VELOCITY                        = mxCreateDoubleMatrix(1,
        number_of_blocks,mxREAL);
    OUT_SHEAR_FORCE                         = mxCreateDoubleMatrix(1,
        number_of_blocks,mxREAL);
    OUT_FRICTION_FORCE                      = mxCreateDoubleMatrix(1,
        number_of_blocks,mxREAL);
    OUT_BLOCK_SPRING_FORCE                  = mxCreateDoubleMatrix(1,
        number_of_blocks,mxREAL);
    OUT_DAMPING_FORCE                       = mxCreateDoubleMatrix(1,
        number_of_blocks,mxREAL);
    OUT_STATIC_FRICTION                     = mxCreateDoubleMatrix(1,
        number_of_blocks,mxREAL);
    OUT_SLIDING                             = mxCreateDoubleMatrix(1,
        number_of_blocks,mxREAL);
    OUT_NEW_LENGTH_STATIC_FRICTION_SPRING = mxCreateDoubleMatrix(1,
        number_of_blocks,mxREAL);
    OUT_DRIVING_FORCE                       = mxCreateDoubleMatrix(1,
        number_of_blocks,mxREAL);
    OUT_ERROR                               = mxCreateDoubleMatrix(1,1,
        mxREAL);
    OUT_T                                   = mxCreateDoubleMatrix(1,1,
        mxREAL);
    OUT_SUM_SLIDING                               = mxCreateDoubleMatrix(1,1,
        mxREAL);

    out_velocity                                  = mxGetPr(OUT_NEW_VELOCITY);
    out_position                                  = mxGetPr(OUT_NEW_POSITION);
    out_shear_force                               = mxGetPr(OUT_SHEAR_FORCE);
    out_friction_force                            = mxGetPr(OUT_FRICTION_FORCE
        );
    out_block_spring_force                        = mxGetPr(
        OUT_BLOCK_SPRING_FORCE);
    out_damping_force                             = mxGetPr(OUT_DAMPING_FORCE)
        ;
    out_static_friction                           = mxGetPr(
        OUT_STATIC_FRICTION);
    out_sliding                                   = mxGetPr(OUT_SLIDING);
    out_length_static_friction_spring             = mxGetPr(
        OUT_NEW_LENGTH_STATIC_FRICTION_SPRING);
    out_driving_force                             = mxGetPr(OUT_DRIVING_FORCE)
        ;
    out_error                                     = mxGetPr(OUT_ERROR);
    out_t                                         = mxGetPr(OUT_T);
    out_sum_sliding                               = mxGetPr(OUT_SUM_SLIDING);

    /* In use inside c-script */
    end_loop                                      = 0;
    out_error[0]                                  = 0;
    first_block                                   = 0;
    sum_sliding_counter                           = 0;
    last_block                                    = number_of_blocks-1;

    tmp_old_velocity                              = (double *)malloc(
        number_of_blocks * sizeof(double));
    tmp_old_position                              = (double *)malloc(
        number_of_blocks * sizeof(double));
    tmp_new_velocity                              = (double *)malloc(
        number_of_blocks * sizeof(double));
    tmp_new_position                              = (double *)malloc(
        number_of_blocks * sizeof(double));
    tmp_old_length_static_friction_spring         = (double *)malloc(
        number_of_blocks * sizeof(double));
    tmp_new_length_static_friction_spring         = (double *)malloc(
        number_of_blocks * sizeof(double));
    tmp_static_friction                           = (double *)malloc(
        number_of_blocks * sizeof(double));
    tmp_driving_force                             = (double *)malloc(
        number_of_blocks * sizeof(double));
    tmp_damping_force                             = (double *)malloc(
        number_of_blocks * sizeof(double));
    tmp_block_spring_force                        = (double *)malloc(
        number_of_blocks * sizeof(double));
    tmp_shear_force                               = (double *)malloc(
        number_of_blocks * sizeof(double));
    tmp_friction_force                            = (double *)malloc(
        number_of_blocks * sizeof(double));
    tmp_sliding                                   = (double *)malloc(
        number_of_blocks * sizeof(double));

    for (i = 0; i < number_of_blocks; i++){
        tmp_old_velocity[i]                           = in_velocity[i];
        tmp_old_position[i]                           = in_position[i];
        tmp_old_length_static_friction_spring[i] =
            in_length_static_friction_spring[i];
        tmp_static_friction[i]                        = in_static_friction[
            i];
        tmp_driving_force[i]                          = 0;
    }

    while (t < t_end && end_loop == 0 && out_error[0] == 0){
        t               = t + 1;
        sum_sliding     = 0;
        /* Calculate the first block */
        tmp_driving_force[first_block]       = driving_spring_stiffness
            *(in_initial_block_positions[first_block] +
            driving_velocity*(dt*t) - tmp_old_position[first_block]);
        tmp_damping_force[first_block]       =
            in_relative_viscous_damping*(tmp_old_velocity[first_block +
             1] - tmp_old_velocity[first_block]);
        tmp_block_spring_force[first_block] =
            in_block_spring_stiffness*( (tmp_old_position[first_block +
             1] - tmp_old_position[first_block]) -
            initial_block_distances);
        tmp_shear_force[first_block]         = tmp_damping_force[
            first_block] + tmp_block_spring_force[first_block] +
            tmp_driving_force[first_block];
        if (tmp_static_friction[first_block] == 1){
            /* Static friction */
            tmp_friction_force[first_block]                      = -
                in_static_friction_spring_stiffness*
                tmp_old_length_static_friction_spring[first_block];
            tmp_new_velocity[first_block]                        =
                tmp_old_velocity[first_block] + (tmp_shear_force[
                first_block] + tmp_friction_force[first_block] )*
                dt_divided_by_mass;
            tmp_new_position[first_block]                        =
                tmp_old_position[first_block] + tmp_new_velocity[
                first_block]*dt;
            tmp_sliding[first_block]                             = 0;
            tmp_static_friction[first_block]                     = 1;
            tmp_new_length_static_friction_spring[first_block] =
                tmp_old_length_static_friction_spring[first_block] +
                tmp_new_position[first_block] - tmp_old_position[
            first_block];
        if (fabs(tmp_shear_force[first_block]) >
            in_static_friction_force[first_block]){
             tmp_sliding[first_block]          = 1;
             sum_sliding                      += 1;
             tmp_static_friction[first_block]  = 0;
        }
    }
    else if (tmp_static_friction[first_block] == 0){
        /* Dynamic friction */
        tmp_friction_force[first_block]                   = -
            in_dynamic_friction_force[first_block]*(
            tmp_old_velocity[first_block]/fabs(tmp_old_velocity[
            first_block]));
        tmp_new_velocity[first_block]                     =
            tmp_old_velocity[first_block] + (tmp_shear_force[
            first_block] + tmp_friction_force[first_block])*
            dt_divided_by_mass;
        tmp_new_position[first_block]                     =
            tmp_old_position[first_block] + tmp_new_velocity[
            first_block]*dt;
        tmp_sliding[first_block]                          = 1;
        sum_sliding                                      += 1;
        tmp_static_friction[first_block]                  = 0;
        tmp_new_length_static_friction_spring[first_block] = 0;
        if (tmp_new_velocity[first_block] < 0){
             tmp_sliding[first_block]                          =
                 0;
             sum_sliding                                      -=
                 1;
             tmp_static_friction[first_block]                  =
                 1;
             tmp_new_length_static_friction_spring[first_block] =
                 tmp_shear_force[first_block]/
                 in_static_friction_spring_stiffness;
        }
    }
    else{
        out_error[0] = 1;
        mexPrintf("ERROR at t = %d [s] for block %d static
            friction = %.1f\n",t,first_block,tmp_static_friction[
            first_block]);
    }

    /* Calculate the last block */
    tmp_driving_force[last_block]      = driving_spring_stiffness
        *(in_initial_block_positions[last_block] + driving_velocity
        *(dt*t) - tmp_old_position[last_block]);
    tmp_block_spring_force[last_block] = in_block_spring_stiffness
        *(initial_block_distances - (tmp_old_position[last_block] -
         tmp_old_position[last_block -1]));
        tmp_damping_force[last_block]       =
            in_relative_viscous_damping*(tmp_old_velocity[last_block -1]
             - tmp_old_velocity[last_block]);
        tmp_shear_force[last_block]         = tmp_damping_force[
            last_block] + tmp_block_spring_force[last_block] +
            tmp_driving_force[last_block];
        if (tmp_static_friction[last_block]  == 1){
             /* Static friction */
             tmp_friction_force[last_block]                    = -
                 in_static_friction_spring_stiffness*
                 tmp_old_length_static_friction_spring[last_block];
             tmp_new_velocity[last_block]                      =
                 tmp_old_velocity[last_block] + (tmp_shear_force[
                 last_block] + tmp_friction_force[last_block])*
                 dt_divided_by_mass;
             tmp_new_position[last_block]                      =
                 tmp_old_position[last_block] + tmp_new_velocity[
                 last_block]*dt;
             tmp_sliding[last_block]                           = 0;
             tmp_static_friction[last_block]                   = 1;
             tmp_new_length_static_friction_spring[last_block] =
                 tmp_old_length_static_friction_spring[last_block] +
                 tmp_new_position[last_block] - tmp_old_position[
                 last_block];
             if (fabs(tmp_shear_force[last_block]) >
                 in_static_friction_force[last_block]){
                  tmp_sliding[last_block]          = 1;
                  sum_sliding                     += 1;
                  tmp_static_friction[last_block] = 0;
             }
        }
        else if (tmp_static_friction[last_block] == 0){
             /* Dynamic friction */
             tmp_friction_force[last_block]                    = -
                 in_dynamic_friction_force[last_block]*(tmp_old_velocity
                 [last_block]/fabs(tmp_old_velocity[last_block]));
             tmp_new_velocity[last_block]                      =
                 tmp_old_velocity[last_block] + (tmp_shear_force[
                 last_block] + tmp_friction_force[last_block])*
                 dt_divided_by_mass;
             tmp_new_position[last_block]                      =
                 tmp_old_position[last_block] + tmp_new_velocity[
                 last_block]*dt;
             tmp_sliding[last_block]                           = 1;
             sum_sliding                                      += 1;
             tmp_static_friction[last_block]                   = 0;
             tmp_new_length_static_friction_spring[last_block] = 0;
             if (tmp_new_velocity[last_block] < 0){
                  tmp_sliding[last_block]                           = 0;
                  sum_sliding                                      -= 1;
                  tmp_static_friction[last_block]                   = 1;
```

```
                tmp_new_length_static_friction_spring[last_block] =
                    tmp_shear_force[last_block]/
                    in_static_friction_spring_stiffness;
            }
        }
        else{
            out_error[0] = 1;
            mexPrintf("ERROR at t = %d [s] for block %d static
                friction = %.1f\n",t,last_block,tmp_static_friction[
                last_block]);
        }

        for (i = 1; i < number_of_blocks - 1; i++){
            /* Calculate the midle blocks */
            tmp_driving_force[i]       = driving_spring_stiffness*(
                in_initial_block_positions[i] + driving_velocity*(dt*t)
                 - tmp_old_position[i]);
            tmp_damping_force[i]       = in_relative_viscous_damping*(
                tmp_old_velocity[i-1] - 2*tmp_old_velocity[i] +
                tmp_old_velocity[i+1]);
            tmp_block_spring_force[i] = in_block_spring_stiffness*(
                tmp_old_position[i+1] - 2*tmp_old_position[i] +
                tmp_old_position[i-1]);
            tmp_shear_force[i]         = tmp_damping_force[i] +
                tmp_block_spring_force[i] + tmp_driving_force[i];
            if (tmp_static_friction[i] == 1){
                /* Static friction */
                tmp_friction_force[i]                      = -
                    in_static_friction_spring_stiffness*
                    tmp_old_length_static_friction_spring[i];
                tmp_new_velocity[i]                        =
                    tmp_old_velocity[i] + (tmp_shear_force[i] +
                    tmp_friction_force[i])*dt_divided_by_mass;
                tmp_new_position[i]                        =
                    tmp_old_position[i] + tmp_new_velocity[i]*dt;
                tmp_sliding[i]                             = 0;
                tmp_static_friction[i]                     = 1;
                tmp_new_length_static_friction_spring[i] =
                    tmp_old_length_static_friction_spring[i] +
                    tmp_new_position[i] - tmp_old_position[i];
                if (fabs(tmp_shear_force[i]) >
                    in_static_friction_force[i]){
                     tmp_sliding[i]          = 1;
                     sum_sliding            += 1;
                     tmp_static_friction[i] = 0;
                }
            }
            else if (tmp_static_friction[i] == 0){
                /* Dynamic friction */
                tmp_friction_force[i]                      = -
                    in_dynamic_friction_force[i]*(tmp_old_velocity[i]/
                            fabs(tmp_old_velocity[i]));
                        tmp_new_velocity[i]                          =
                            tmp_old_velocity[i] + (tmp_shear_force[i] +
                            tmp_friction_force[i])*dt_divided_by_mass;
                        tmp_new_position[i]                          =
                            tmp_old_position[i] + tmp_new_velocity[i]*dt;
                        tmp_sliding[i]                               = 1;
                        sum_sliding                                 += 1;
                        tmp_static_friction[i]                       = 0;
                        tmp_new_length_static_friction_spring[i]     = 0;
                        if (tmp_new_velocity[i] < 0){
                             tmp_sliding[i]                          = 0;
                             sum_sliding                            -= 1;
                             tmp_static_friction[i]                  = 1;
                             tmp_new_length_static_friction_spring[i] =
                                 tmp_shear_force[i]/
                                 in_static_friction_spring_stiffness;
                        }
                   }
                   else{
                        out_error[0] = 1;
                        mexPrintf("ERROR at t = %d [s] for block %d static
                            friction = %.1f\n",t,i,tmp_static_friction[i]);
                   }
             }
             for (j = 0; j < number_of_blocks;j++){
                   tmp_old_position[j] = tmp_new_position[j];
                   tmp_old_velocity[j] = tmp_new_velocity[j];
                   tmp_old_length_static_friction_spring[j] =
                       tmp_new_length_static_friction_spring[j];
             }
             if (((sum_sliding > 0) && (sum_sliding != in_sum_sliding)) ||
                 ((sum_sliding == 0) && (in_sum_sliding > 0))){
                   end_loop        = 1;
             }
       }

       *out_sum_sliding = sum_sliding;
       *out_t           = t;
       for (k = 0; k < number_of_blocks;k++){
             out_static_friction[k]                   = tmp_static_friction[k];
             out_position[k]                          = tmp_new_position[k];
             out_velocity[k]                          = tmp_new_velocity[k];
             out_length_static_friction_spring[k] =
                 tmp_new_length_static_friction_spring[k];
             out_driving_force[k]                     = tmp_driving_force[k];
             out_damping_force[k]                     = tmp_damping_force[k];
             out_block_spring_force[k]                = tmp_block_spring_force[
                 k];
             out_shear_force[k]                       = tmp_shear_force[k];
             out_friction_force[k]                    = tmp_friction_force[k];
```

```
        out_sliding[k]                          = tmp_sliding[k];
    }

    free(tmp_old_velocity);
    free(tmp_old_position);
    free(tmp_old_length_static_friction_spring);
    free(tmp_new_velocity);
    free(tmp_new_position);
    free(tmp_new_length_static_friction_spring);
    free(tmp_static_friction);
    free(tmp_driving_force);
    free(tmp_damping_force);
    free(tmp_block_spring_force);
    free(tmp_shear_force);
    free(tmp_friction_force);
    free(tmp_sliding);
    return;
}
```

B.5 The C code for the unloading model

```
#include <math.h>
#include "mex.h"

/* Input Arguments */
#define IN_POSITION                                 prhs[0]
#define IN_VELOCITY                                 prhs[1]
#define IN_LENGTH_STATIC_FRICTION_SPRING            prhs[2]

#define IN_STATIC_FRICTION                          prhs[3]
#define IN_INITIAL_POSITIONS                        prhs[4]
#define IN_NORMAL_FORCE_PER_BLOCK                   prhs[5]

#define IN_DRIVING_MATRIX                           prhs[6]

#define IN_RELATIVE_VISCOUS_DAMPING                 prhs[7]
#define IN_BLOCK_SPRING_STIFFNESS                   prhs[8]
#define IN_STATIC_FRICTION_SPRING_STIFFNESS         prhs[9]

#define IN_DT_DIVIDED_BY_MASS                       prhs[10]
#define IN_INITIAL_BLOCK_DISTANCES                  prhs[11]
#define IN_DRIVING_SPRING_STIFFNESS                 prhs[12]
#define IN_DRIVING_VELOCITY                         prhs[13]
#define IN_NUMBER_OF_BLOCKS                         prhs[14]

#define IN_T_NORMAL_FORCE                           prhs[15]
#define IN_T_END                                    prhs[16]
#define IN_T                                        prhs[17]
#define IN_DT                                       prhs[18]

#define IN_SUM_SLIDING                              prhs[19]
#define IN_STATIC_FRICTION_COEFFICIENT              prhs[20]
#define IN_DYNAMIC_FRICTION_COEFFICIENT             prhs[21]

#define IN_GAMMA                                    prhs[22]
#define IN_GAMMA_PROFILE_SLOPE                      prhs[23]
#define IN_GAMMA_PROFILE_TIME                       prhs[24]
#define IN_GAMMA_COUNTER                            prhs[25]

#define IN_POSITION_DRIVING_SPRING                  prhs[26]
#define IN_CONST_DRIVING_FORCE_TIME                 prhs[27]

/* Output Arguments */
#define OUT_NEW_POSITION                             plhs[0]
#define OUT_NEW_VELOCITY                             plhs[1]
#define OUT_SHEAR_FORCE                              plhs[2]
#define OUT_FRICTION_FORCE                           plhs[3]
#define OUT_BLOCK_SPRING_FORCE                       plhs[4]
#define OUT_DAMPING_FORCE                            plhs[5]
```

```
#define OUT_STATIC_FRICTION                        plhs[6]
#define OUT_SLIDING                                plhs[7]
#define OUT_NEW_LENGTH_STATIC_FRICTION_SPRING      plhs[8]
#define OUT_DRIVING_FORCE                          plhs[9]
#define OUT_ERROR                                  plhs[10]
#define OUT_T                                      plhs[11]
#define OUT_T_NORMAL_FORCE                         plhs[12]
#define OUT_SUM_SLIDING                            plhs[13]

#define OUT_STATIC_FRICTION_FORCE                  plhs[14]
#define OUT_DYNAMIC_FRICTION_FORCE                 plhs[15]
#define OUT_NORMAL_FORCE_PER_BLOCK                 plhs[16]
#define OUT_GAMMA_COUNTER                          plhs[17]
#define OUT_GAMMA                                  plhs[18]

#define OUT_POSITION_DRIVING_SPRING                plhs[19]

 void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray*
    prhs[]){
     /* Input */
     double *in_velocity, *in_position, *
        in_length_static_friction_spring;
     double *in_static_friction, *in_initial_block_positions;
     double *in_normal_force_per_block;
     double *in_driving_matrix;
     double in_relative_viscous_damping;
     double in_block_spring_stiffness,
        in_static_friction_spring_stiffness;
     double dt_divided_by_mass, initial_block_distances;
     double driving_spring_stiffness, driving_velocity,
        number_of_blocks;
     double dt;
     double t, t_end, t_normal_force;
     double in_sum_sliding;
     double in_static_friction_coefficient,
        in_dynamic_friction_coefficient;
     double in_gamma;
     double *in_gamma_profile_slope, *in_gamma_profile_time;
     double in_gamma_counter;
     double *in_position_driving_spring;
     double in_const_driving_force_time;

     /* Output */
     double *out_position, *out_velocity;
     double *out_shear_force, *out_friction_force, *
        out_block_spring_force, *out_damping_force;
     double *out_static_friction, *out_sliding, *
        out_length_static_friction_spring;
     double *out_driving_force;
     double *out_error;
     double *out_t;
    double *out_t_normal_force;
    double *out_sum_sliding;
    double *out_static_friction_force;
    double *out_dynamic_friction_force;
    double *out_normal_force_per_block;
    double *out_gamma_counter;
    double *out_gamma;
    double *out_position_driving_spring;

    /* In use inside c-script */
    int i, j, k;
    int first_block, last_block;
    double tmp_gamma_counter;
    double gamma_from_gamma_profile;
    double end_loop;
    double sum_sliding;
    double sum_sliding_counter;
    double tmp_t_normal_force;
    double* tmp_old_position;
    double* tmp_old_velocity;
    double* tmp_old_length_static_friction_spring;
    double* tmp_new_position;
    double* tmp_new_velocity;
    double* tmp_new_length_static_friction_spring;
    double* tmp_static_friction;
    double* tmp_driving_force;
    double* tmp_damping_force;
    double* tmp_block_spring_force;
    double* tmp_shear_force;
    double* tmp_friction_force;
    double* tmp_sliding;
    double* tmp_static_friction_force;
    double* tmp_dynamic_friction_force;
    double* tmp_normal_force_per_block;
    double* tmp_position_driving_spring;


    /* Input */
    in_velocity                          = mxGetPr(IN_VELOCITY);
    in_position                          = mxGetPr(IN_POSITION);
    in_length_static_friction_spring     = mxGetPr(
        IN_LENGTH_STATIC_FRICTION_SPRING);
    in_static_friction                   = mxGetPr(IN_STATIC_FRICTION
        );
    in_initial_block_positions           = mxGetPr(
        IN_INITIAL_POSITIONS);
    in_normal_force_per_block            = mxGetPr(
        IN_NORMAL_FORCE_PER_BLOCK);
    in_driving_matrix                    = mxGetPr(IN_DRIVING_MATRIX)
        ;
    in_gamma_profile_slope                  = mxGetPr(
        IN_GAMMA_PROFILE_SLOPE);
    in_gamma_profile_time                   = mxGetPr(
        IN_GAMMA_PROFILE_TIME);
    in_position_driving_spring              = mxGetPr(
        IN_POSITION_DRIVING_SPRING);

    in_relative_viscous_damping             = mxGetScalar(
        IN_RELATIVE_VISCOUS_DAMPING);
    in_block_spring_stiffness               = mxGetScalar(
        IN_BLOCK_SPRING_STIFFNESS);
    in_static_friction_spring_stiffness     = mxGetScalar(
        IN_STATIC_FRICTION_SPRING_STIFFNESS);
    dt_divided_by_mass                      = mxGetScalar(
        IN_DT_DIVIDED_BY_MASS);
    initial_block_distances                 = mxGetScalar(
        IN_INITIAL_BLOCK_DISTANCES);
    driving_spring_stiffness                = mxGetScalar(
        IN_DRIVING_SPRING_STIFFNESS);
    driving_velocity                        = mxGetScalar(
        IN_DRIVING_VELOCITY);
    number_of_blocks                        = mxGetScalar(
        IN_NUMBER_OF_BLOCKS);
    t_normal_force                          = mxGetScalar(
        IN_T_NORMAL_FORCE);
    t_end                                   = mxGetScalar(IN_T_END);
    t                                       = mxGetScalar(IN_T);
    dt                                      = mxGetScalar(IN_DT);
    in_sum_sliding                          = mxGetScalar(IN_SUM_SLIDING
        );
    in_static_friction_coefficient          = mxGetScalar(
        IN_STATIC_FRICTION_COEFFICIENT);
    in_dynamic_friction_coefficient         = mxGetScalar(
        IN_DYNAMIC_FRICTION_COEFFICIENT);
    in_gamma                                = mxGetScalar(IN_GAMMA);
    in_gamma_counter                        = mxGetScalar(
        IN_GAMMA_COUNTER);
    in_const_driving_force_time             = mxGetScalar(
        IN_CONST_DRIVING_FORCE_TIME);

    /* Output */
    OUT_NEW_POSITION                        = mxCreateDoubleMatrix(1,
        number_of_blocks,mxREAL);
    OUT_NEW_VELOCITY                        = mxCreateDoubleMatrix(1,
        number_of_blocks,mxREAL);
    OUT_SHEAR_FORCE                         = mxCreateDoubleMatrix(1,
        number_of_blocks,mxREAL);
    OUT_FRICTION_FORCE                      = mxCreateDoubleMatrix(1,
        number_of_blocks,mxREAL);
    OUT_BLOCK_SPRING_FORCE                  = mxCreateDoubleMatrix(1,
        number_of_blocks,mxREAL);
    OUT_DAMPING_FORCE                      = mxCreateDoubleMatrix(1,
         number_of_blocks,mxREAL);
    OUT_STATIC_FRICTION                    = mxCreateDoubleMatrix(1,
         number_of_blocks,mxREAL);
    OUT_SLIDING                            = mxCreateDoubleMatrix(1,
         number_of_blocks,mxREAL);
    OUT_NEW_LENGTH_STATIC_FRICTION_SPRING  = mxCreateDoubleMatrix(1,
         number_of_blocks,mxREAL);
    OUT_DRIVING_FORCE                      = mxCreateDoubleMatrix(1,
         number_of_blocks,mxREAL);
    OUT_STATIC_FRICTION_FORCE              = mxCreateDoubleMatrix(1,
         number_of_blocks,mxREAL);
    OUT_DYNAMIC_FRICTION_FORCE             = mxCreateDoubleMatrix(1,
         number_of_blocks,mxREAL);
    OUT_NORMAL_FORCE_PER_BLOCK             = mxCreateDoubleMatrix(1,
         number_of_blocks,mxREAL);
    OUT_POSITION_DRIVING_SPRING            = mxCreateDoubleMatrix(1,
         number_of_blocks,mxREAL);
    OUT_ERROR                              = mxCreateDoubleMatrix(1,1,
         mxREAL);
    OUT_T_NORMAL_FORCE                     = mxCreateDoubleMatrix(1,1,
         mxREAL);
    OUT_T                                  = mxCreateDoubleMatrix(1,1,
         mxREAL);
    OUT_SUM_SLIDING                        = mxCreateDoubleMatrix(1,1,
         mxREAL);
    OUT_GAMMA_COUNTER                      = mxCreateDoubleMatrix(1,1,
         mxREAL);
    OUT_GAMMA                              = mxCreateDoubleMatrix(1,1,
         mxREAL);

    out_velocity                           = mxGetPr(OUT_NEW_VELOCITY);
    out_position                           = mxGetPr(OUT_NEW_POSITION);
    out_shear_force                        = mxGetPr(OUT_SHEAR_FORCE);
    out_friction_force                     = mxGetPr(OUT_FRICTION_FORCE
         );
    out_block_spring_force                 = mxGetPr(
         OUT_BLOCK_SPRING_FORCE);
    out_damping_force                      = mxGetPr(OUT_DAMPING_FORCE)
         ;
    out_static_friction                    = mxGetPr(
         OUT_STATIC_FRICTION);
    out_sliding                            = mxGetPr(OUT_SLIDING);
    out_length_static_friction_spring      = mxGetPr(
         OUT_NEW_LENGTH_STATIC_FRICTION_SPRING);
    out_driving_force                      = mxGetPr(OUT_DRIVING_FORCE)
         ;
    out_static_friction_force              = mxGetPr(
         OUT_STATIC_FRICTION_FORCE);
    out_dynamic_friction_force             = mxGetPr(
         OUT_DYNAMIC_FRICTION_FORCE);
    out_normal_force_per_block            = mxGetPr(
        OUT_NORMAL_FORCE_PER_BLOCK);
    out_position_driving_spring           = mxGetPr(
        OUT_POSITION_DRIVING_SPRING);

    out_error                             = mxGetPr(OUT_ERROR);
    out_t_normal_force                    = mxGetPr(OUT_T_NORMAL_FORCE
        );
    out_t                                 = mxGetPr(OUT_T);
    out_sum_sliding                       = mxGetPr(OUT_SUM_SLIDING);
    out_gamma_counter                     = mxGetPr(OUT_GAMMA_COUNTER)
        ;
    out_gamma                             = mxGetPr(OUT_GAMMA);

    /* In use inside c-script */
    end_loop                              = 0;
    out_error[0]                          = 0;
    first_block                           = 0;
    sum_sliding_counter                   = 0;
    tmp_gamma_counter                     = in_gamma_counter;
    gamma_from_gamma_profile              = in_gamma;
    last_block                            = number_of_blocks-1;

    tmp_old_velocity                      = (double *)malloc(
        number_of_blocks * sizeof(double));
    tmp_old_position                      = (double *)malloc(
        number_of_blocks * sizeof(double));
    tmp_new_velocity                      = (double *)malloc(
        number_of_blocks * sizeof(double));
    tmp_new_position                      = (double *)malloc(
        number_of_blocks * sizeof(double));
    tmp_old_length_static_friction_spring = (double *)malloc(
        number_of_blocks * sizeof(double));
    tmp_new_length_static_friction_spring = (double *)malloc(
        number_of_blocks * sizeof(double));
    tmp_static_friction                   = (double *)malloc(
        number_of_blocks * sizeof(double));
    tmp_driving_force                     = (double *)malloc(
        number_of_blocks * sizeof(double));
    tmp_damping_force                     = (double *)malloc(
        number_of_blocks * sizeof(double));
    tmp_block_spring_force                = (double *)malloc(
        number_of_blocks * sizeof(double));
    tmp_shear_force                       = (double *)malloc(
        number_of_blocks * sizeof(double));
    tmp_friction_force                    = (double *)malloc(
        number_of_blocks * sizeof(double));
    tmp_sliding                           = (double *)malloc(
        number_of_blocks * sizeof(double));
    tmp_static_friction_force             = (double *)malloc(
        number_of_blocks * sizeof(double));
    tmp_dynamic_friction_force             = (double *)malloc(
        number_of_blocks * sizeof(double));
    tmp_normal_force_per_block             = (double *)malloc(
        number_of_blocks * sizeof(double));
    tmp_position_driving_spring            = (double *)malloc(
        number_of_blocks * sizeof(double));

    for (i = 0; i < number_of_blocks; i++){
        tmp_old_velocity[i]                       = in_velocity[i];
        tmp_old_position[i]                       = in_position[i];
        tmp_old_length_static_friction_spring[i]  =
            in_length_static_friction_spring[i];
        tmp_static_friction[i]                    = in_static_friction[
            i];
        tmp_normal_force_per_block[i]             =
            in_normal_force_per_block[i];
        tmp_driving_force[i]                      = 0;
        tmp_position_driving_spring[i]            =
            in_position_driving_spring[i];
    }

    while (t < t_end && end_loop == 0 && out_error[0] == 0){
        t           = t + 1;
        sum_sliding = 0;
        if (t == (long int)in_gamma_profile_time[(int)
            tmp_gamma_counter]){
            gamma_from_gamma_profile = in_gamma_profile_slope[(int)
                tmp_gamma_counter];
            tmp_gamma_counter += 1;
        }

        for (i = 0; i < number_of_blocks; i++){
            tmp_normal_force_per_block[i] += (gamma_from_gamma_profile
                *dt)/number_of_blocks;
            tmp_static_friction_force[i]    =
                in_static_friction_coefficient*
                tmp_normal_force_per_block[i];
            tmp_dynamic_friction_force[i]   =
                in_dynamic_friction_coefficient*
                tmp_normal_force_per_block[i];
        }
        if (t < (long int)in_const_driving_force_time){
            tmp_position_driving_spring[first_block] =
                in_initial_block_positions[first_block] +
                driving_velocity*(dt*t);
            tmp_position_driving_spring[last_block]   =
                in_initial_block_positions[last_block] +
                driving_velocity*(dt*t);
            tmp_driving_force[first_block] = in_driving_matrix[0]*
                driving_spring_stiffness*(tmp_position_driving_spring[
                first_block] - tmp_old_position[first_block]);
        tmp_driving_force[last_block]  = in_driving_matrix[1]*
            driving_spring_stiffness*(tmp_position_driving_spring[
            last_block] - tmp_old_position[last_block]);
    }else if (t >= (long int)in_const_driving_force_time){
        tmp_position_driving_spring[first_block] =
            in_initial_block_positions[first_block] +
            driving_velocity*(dt*in_const_driving_force_time);
        tmp_position_driving_spring[last_block]  =
            in_initial_block_positions[last_block] +
            driving_velocity*(dt*in_const_driving_force_time);
        tmp_driving_force[first_block] = in_driving_matrix[0]*
            driving_spring_stiffness*(tmp_position_driving_spring[
            first_block] - tmp_old_position[first_block]);
        tmp_driving_force[last_block]  = in_driving_matrix[1]*
            driving_spring_stiffness*(tmp_position_driving_spring[
            last_block] - tmp_old_position[last_block]);
    }

    /* Calculate the first block */
    tmp_damping_force[first_block]      =
        in_relative_viscous_damping*(tmp_old_velocity[first_block +
         1] - tmp_old_velocity[first_block]);
    tmp_block_spring_force[first_block] =
        in_block_spring_stiffness*( (tmp_old_position[first_block +
         1] - tmp_old_position[first_block]) -
        initial_block_distances);
    tmp_shear_force[first_block]        = tmp_damping_force[
        first_block] + tmp_block_spring_force[first_block] +
        tmp_driving_force[first_block];
    if (tmp_static_friction[first_block] == 1){
        /* Static friction */
        tmp_friction_force[first_block]                   = -
            in_static_friction_spring_stiffness*
            tmp_old_length_static_friction_spring[first_block];
        tmp_new_velocity[first_block]                     =
            tmp_old_velocity[first_block] + (tmp_shear_force[
            first_block] + tmp_friction_force[first_block] )*
            dt_divided_by_mass;
        tmp_new_position[first_block]                     =
            tmp_old_position[first_block] + tmp_new_velocity[
            first_block]*dt;
        tmp_sliding[first_block]                          = 0;
        tmp_static_friction[first_block]                  = 1;
        tmp_new_length_static_friction_spring[first_block] =
            tmp_old_length_static_friction_spring[first_block] +
            tmp_new_position[first_block] - tmp_old_position[
            first_block];
        if (fabs(tmp_shear_force[first_block]) >
            tmp_static_friction_force[first_block]){
            /*tmp_sliding[first_block]         = 1;*/
            /*sum_sliding                     += 1;*/
```

```
                tmp_static_friction[first_block]   = 0;
            }
        }else if (tmp_static_friction[first_block] == 0){
            /* Dynamic friction */
            tmp_friction_force[first_block]                    = -
                tmp_dynamic_friction_force[first_block]*(
                tmp_old_velocity[first_block]/fabs(tmp_old_velocity[
                first_block]));
            tmp_new_velocity[first_block]                      =
                tmp_old_velocity[first_block] + (tmp_shear_force[
                first_block] + tmp_friction_force[first_block])*
                dt_divided_by_mass;
            tmp_new_position[first_block]                      =
                tmp_old_position[first_block] + tmp_new_velocity[
                first_block]*dt;
            tmp_sliding[first_block]                           = 1;
            sum_sliding                                       += 1;
            tmp_static_friction[first_block]                   = 0;
            tmp_new_length_static_friction_spring[first_block] = 0;
            if (tmp_new_velocity[first_block] < 0){
                /*tmp_sliding[first_block]                         =
                    0;*/
                /*sum_sliding                                     -=
                    1;*/
                tmp_static_friction[first_block]                 =
                    1;
                tmp_new_length_static_friction_spring[first_block] =
                    tmp_shear_force[first_block]/
                    in_static_friction_spring_stiffness;
            }
        }else{
            out_error[0] = 1;
            mexPrintf("ERROR at t = %d [s] for block %d static
                friction = %.1f\n",t,first_block,tmp_static_friction[
                first_block]);
        }

        /* Calculate the last block */
        tmp_block_spring_force[last_block] = in_block_spring_stiffness
            *(initial_block_distances - (tmp_old_position[last_block] -
             tmp_old_position[last_block-1]));
        tmp_damping_force[last_block]      =
            in_relative_viscous_damping*(tmp_old_velocity[last_block-1]
             - tmp_old_velocity[last_block]);
        tmp_shear_force[last_block]         = tmp_damping_force[
            last_block] + tmp_block_spring_force[last_block] +
            tmp_driving_force[last_block];
        if (tmp_static_friction[last_block]  == 1){
            /* Static friction */
            tmp_friction_force[last_block]                     = -
                in_static_friction_spring_stiffness*
                tmp_old_length_static_friction_spring[last_block];
            tmp_new_velocity[last_block]                            =
                tmp_old_velocity[last_block] + (tmp_shear_force[
                last_block] + tmp_friction_force[last_block])*
                dt_divided_by_mass;
            tmp_new_position[last_block]                            =
                tmp_old_position[last_block] + tmp_new_velocity[
                last_block]*dt;
            tmp_sliding[last_block]                                 = 0;
            tmp_static_friction[last_block]                         = 1;
            tmp_new_length_static_friction_spring[last_block] =
                tmp_old_length_static_friction_spring[last_block] +
                tmp_new_position[last_block] - tmp_old_position[
                last_block];
            if (fabs(tmp_shear_force[last_block]) >
                tmp_static_friction_force[last_block]){
                 /*tmp_sliding[last_block]            = 1;*/
                 /*sum_sliding                       += 1;*/
                 tmp_static_friction[last_block] = 0;
            }
        }else if (tmp_static_friction[last_block] == 0){
            /* Dynamic friction */
            tmp_friction_force[last_block]                          = -
                tmp_dynamic_friction_force[last_block]*(
                tmp_old_velocity[last_block]/fabs(tmp_old_velocity[
                last_block]));
            tmp_new_velocity[last_block]                            =
                tmp_old_velocity[last_block] + (tmp_shear_force[
                last_block] + tmp_friction_force[last_block])*
                dt_divided_by_mass;
            tmp_new_position[last_block]                            =
                tmp_old_position[last_block] + tmp_new_velocity[
                last_block]*dt;
            tmp_sliding[last_block]                                 = 1;
            sum_sliding                                            += 1;
            tmp_static_friction[last_block]                         = 0;
            tmp_new_length_static_friction_spring[last_block] = 0;
            if (tmp_new_velocity[last_block] < 0){
                 /*tmp_sliding[last_block]                                =
                     0;*/
                 /*sum_sliding                                           -=
                     1;*/
                 tmp_static_friction[last_block]                          = 1;
                 tmp_new_length_static_friction_spring[last_block] =
                     tmp_shear_force[last_block]/
                     in_static_friction_spring_stiffness;
            }
        }else{
            out_error[0] = 1;
            mexPrintf("ERROR at t = %d [s] for block %d static
                friction = %.1f\n",t,last_block,tmp_static_friction[
```

```c
                    last_block]);
            }

            for (i = 1; i < number_of_blocks - 1; i++){
                /* Calculate the midle blocks */
                tmp_damping_force[i]      = in_relative_viscous_damping*(
                    tmp_old_velocity[i-1] - 2*tmp_old_velocity[i] +
                    tmp_old_velocity[i+1]);
                tmp_block_spring_force[i] = in_block_spring_stiffness*(
                    tmp_old_position[i+1] - 2*tmp_old_position[i] +
                    tmp_old_position[i-1]);
                tmp_shear_force[i]        = tmp_damping_force[i] +
                    tmp_block_spring_force[i];
                if (tmp_static_friction[i] == 1){
                    /* Static friction */
                    tmp_friction_force[i]                       = -
                        in_static_friction_spring_stiffness*
                        tmp_old_length_static_friction_spring[i];
                    tmp_new_velocity[i]                         =
                        tmp_old_velocity[i] + (tmp_shear_force[i] +
                        tmp_friction_force[i])*dt_divided_by_mass;
                    tmp_new_position[i]                         =
                        tmp_old_position[i] + tmp_new_velocity[i]*dt;
                    tmp_sliding[i]                              = 0;
                    tmp_static_friction[i]                      = 1;
                    tmp_new_length_static_friction_spring[i] =
                        tmp_old_length_static_friction_spring[i] +
                        tmp_new_position[i] - tmp_old_position[i];
                    if (fabs(tmp_shear_force[i]) >
                        tmp_static_friction_force[i]){
                         /*tmp_sliding[i]          = 1;*/
                         /*sum_sliding             += 1;*/
                         tmp_static_friction[i] = 0;
                    }
                }else if (tmp_static_friction[i] == 0){
                    /* Dynamic friction */
                    tmp_friction_force[i]                       = -
                        tmp_dynamic_friction_force[i]*(tmp_old_velocity[i]/
                        fabs(tmp_old_velocity[i]));
                    tmp_new_velocity[i]                         =
                        tmp_old_velocity[i] + (tmp_shear_force[i] +
                        tmp_friction_force[i])*dt_divided_by_mass;
                    tmp_new_position[i]                         =
                        tmp_old_position[i] + tmp_new_velocity[i]*dt;
                    tmp_sliding[i]                              = 1;
                    sum_sliding                                += 1;
                    tmp_static_friction[i]                      = 0;
                    tmp_new_length_static_friction_spring[i] = 0;
                    if (tmp_new_velocity[i] < 0){
                         /*tmp_sliding[i]                          = 0;*/
                         /*sum_sliding                            -= 1;*/
                    tmp_static_friction[i]                  = 1;
                    tmp_new_length_static_friction_spring[i] =
                        tmp_shear_force[i]/
                        in_static_friction_spring_stiffness;
                }
            }else{
                out_error[0] = 1;
                mexPrintf("ERROR at t = %d [s] for block %d static
                    friction = %.1f\n",t,i,tmp_static_friction[i]);
            }
        }
        for (j = 0; j < number_of_blocks;j++){
            tmp_old_position[j] = tmp_new_position[j];
            tmp_old_velocity[j] = tmp_new_velocity[j];
            tmp_old_length_static_friction_spring[j] =
                tmp_new_length_static_friction_spring[j];
        }
        if (((sum_sliding > 0) && (sum_sliding != in_sum_sliding)) ||
            ((sum_sliding == 0) && (in_sum_sliding > 0))){
            end_loop         = 1;
        }
    }

    *out_sum_sliding    = sum_sliding;
    *out_t              = t;
    *out_gamma_counter  = tmp_gamma_counter;
    *out_gamma          = gamma_from_gamma_profile;

    for (k = 0; k < number_of_blocks;k++){
        out_static_friction[k]                  = tmp_static_friction[k];
        out_position[k]                         = tmp_new_position[k];
        out_velocity[k]                         = tmp_new_velocity[k];
        out_length_static_friction_spring[k] =
            tmp_new_length_static_friction_spring[k];
        out_driving_force[k]                    = tmp_driving_force[k];
        out_damping_force[k]                    = tmp_damping_force[k];
        out_block_spring_force[k]               = tmp_block_spring_force[
            k];
        out_shear_force[k]                      = tmp_shear_force[k];
        out_friction_force[k]                   = tmp_friction_force[k];
        out_sliding[k]                          = tmp_sliding[k];
        out_static_friction_force[k]            =
            tmp_static_friction_force[k];
        out_dynamic_friction_force[k]           =
            tmp_dynamic_friction_force[k];
        out_normal_force_per_block[k]           =
            tmp_normal_force_per_block[k];
        out_position_driving_spring[k]          =
            tmp_position_driving_spring[k];
    }
    free(tmp_old_velocity);
    free(tmp_old_position);
    free(tmp_old_length_static_friction_spring);
    free(tmp_new_velocity);
    free(tmp_new_position);
    free(tmp_new_length_static_friction_spring);
    free(tmp_static_friction);
    free(tmp_driving_force);
    free(tmp_damping_force);
    free(tmp_block_spring_force);
    free(tmp_shear_force);
    free(tmp_friction_force);
    free(tmp_sliding);
    free(tmp_static_friction_force);
    free(tmp_dynamic_friction_force);
    free(tmp_normal_force_per_block);
    free(tmp_position_driving_spring);
    return;
}
```

Bibliography

[1] Berni J Alder and TE Wainwright. "Studies in molecular dynamics. I. General method". In: *The Journal of Chemical Physics* 31 (1959), p. 459.

[2] D.S. Amundsen. "Modelling the onset of dynamic friction: A study of rupture velocities". MA thesis. Oslo: Univeristy of Oslo, 2011.

[3] D.S. Amundsen et al. "1D model of precursors to frictional stick-slip motion allowing for robust comparison with experiments". In: *Tribology Letters* (2012), pp. 1–13.

[4] *Atomic force microscopy Block diagram*. http://en.wikipedia.org/wiki/Atomic_force_microscopy. Accessed: 02.03.2014.

[5] T. Baumberger and C. Caroli. "Solid friction from stick–slip down to pinning and aging". In: *Advances in Physics* 55.3-4 (2006), pp. 279–348.

[6] O Ben-David, G Cohen, and J Fineberg. "Short-time dynamics of frictional strength in dry friction". In: *Tribology letters* 39.3 (2010), pp. 235–245.

[7] O. Ben-David, G. Cohen, and J. Fineberg. "The dynamics of the onset of frictional slip". In: *Science* 330.6001 (2010), pp. 211–214.

[8] O. Ben-David, S.M. Rubinstein, and J. Fineberg. "Slip-stick and the evolution of frictional strength". In: *Nature* 463.7277 (2010), pp. 76–79.

[9] M. L. Boas. *Mathematical Methods in the Physical Sciences*. third edition. Hoboken, NJ: John Wiley & Sons Inc., 2006.

[10] OM Braun, I. Barel, and M. Urbakh. "Dynamics of transition from static to kinetic friction". In: *Physical Review Letters* 103.19 (2009), p. 194301.

[11] R Burridge and Leon Knopoff. "Model and theoretical seismicity". In: *Bulletin of the Seismological Society of America* 57.3 (1967), pp. 341–371.

[12] J. C. Butcher. *Numerical Methods for Ordinary Differential equations*. second edition. John Wiley & Sons, Ltd, 2008.

[13] Jean M Carlson, James S Langer, and Bruce E Shaw. "Dynamics of earthquake faults". In: *Reviews of Modern Physics* 66.2 (1994), p. 657.

[14] Duncan Dowson. *History of tribology*. second edition. London and Bury St Edmunds, UK: Professional Engineering Publishing Limited, 1998.

[15] *Euler method Truncation error*. en.wikipedia.org/wiki/Euler's_method. Accessed: 03.03.2014.

[16] L Knopoff, JO Mouton, and R Burridge. "The Dynamics of a one-dimensional Fault in the Presence of FrictiontâĂă". In: *Geophysical Journal of the Royal Astronomical Society* 35.1-3 (1973), pp. 169–184.

[17] S. Maegawa, A. Suzuki, and K. Nakano. "Precursors of global slip in a longitudinal line contact under non-uniform normal loading". In: *Tribology Letters* 38.3 (2010), pp. 313–323.

[18] Gregory C McLaskey et al. "Fault healing promotes high-frequency earthquakes in laboratory experiments and on natural faults". In: *Nature* 491.7422 (2012), pp. 101–104.

[19] Nicholas Metropolis and Stanislaw Ulam. "The monte carlo method". In: *Journal of the American statistical association* 44.247 (1949), pp. 335–341.

[20] *Poly(methyl methacrylate) Young's modulus*. http://en.wikipedia.org/wiki/Poly%28methyl_methacrylate%29. Accessed: 02.03.2014.

[21] H. W. Press et al. *Numerical Recipes*. third edition. New York: Cambridge Pniversity press, 2007.

[22] Shmuel M Rubinstein et al. "Crack-like processes governing the onset of frictional slip". In: *International journal of fracture* 140.1-4 (2006), pp. 201–212.

[23] S.M. Rubinstein, G. Cohen, and J. Fineberg. "Detachment fronts and the onset of dynamic friction". In: *Nature* 430.7003 (2004), pp. 1005–1009.

[24] SM Rubinstein, G Cohen, and J Fineberg. "Dynamics of precursors to frictional sliding". In: *Physical review letters* 98.22 (2007), p. 226103.

[25] SM Rubinstein, G Cohen, and J Fineberg. "Visualizing stick–slip: experimental observations of processes governing the nucleation of frictional sliding". In: *Journal of Physics D: Applied Physics* 42.21 (2009), p. 214016.

[26] PA Selvadurai, SD Glaser, et al. "Direct measurement of contact area and seismic stress along a sliding interface". In: *46th US Rock Mechanics/Geomechanics Symposium*. American Rock Mechanics Association. 2012.

[27] J. Trømborg. "Modelling the onset of dynamic friction: Importace of the vertical dimension". MA thesis. Oslo: Univeristy of Oslo, 2011.