BACHELOR

Real time analysis of DNA force-distance curves

Boef, den, D.

Award date: 2017

Link to publication
Real time analysis of DNA force-distance curves

David den Boef

July 12, 2017

Abstract

We tried to find a way to automatically determine if a single DNA molecule has been attached to the beads in an dual optical trap experiment. This was done by looking at the force-distance curve, and using a classifier based on kernel density estimation. We looked at the differences between the naive approach of simply using the raw FD-data compared to the more complex approach of fitting a worm-like chain model to the curve and using those result for classification. We considered three cases: no molecule attached, 1 molecule attached and 2 molecules attached. We found that using the worm-like chain model for fitting does not improve the classification. It should be noted that we did not use actual measured data for the most part. Simulation were performed to obtain the FD-curves. We conclude that it is not necessary to use a physical model for classification, using the raw data works just as well, if not even better. So it is probably better to improve upon the classification using the raw data compared to using a worm-like chain. Which has the additional advantage of being applicable to a wider range of physical systems.
## Contents

1 Introduction 3  
1.1 Outline 3  

2 Experimental setup 5  
2.1 Optical Tweezers 5  
2.2 Catching DNA 6  

3 Theory 7  
3.1 Worm-Like Chain 7  
3.2 Kernel density estimates 8  
3.2.1 Classification using KDEs 9  
3.2.2 Automatic bandwidth selection 10  
3.3 Logistic regression 11  

4 Implementation 13  
4.1 Curve generation 13  
4.2 Curve fitting 13  
4.3 Classification 14  

5 Results 16  

6 Outlook and Conclusion 20  

References 21  

A Protocol Description 23  
A.1 IPC protocol 23  

B Code listing 24  
B.1 Analysis Module 24  
B.2 Experiment Simulator 42  
B.3 KDE Creator 65  
B.4 Utils 76
1 Introduction

DNA is a biomolecule which contains all the genetic information of a living organisms. Like a computer, which stores its information in a binary fashion, DNA uses four building blocks called nucleotides to store its information. These are ATCG. Understanding of DNA is of great interest to the medical community, since errors in DNA are linked to various diseases like cancer \cite{10}. In the last 25 years, it has been made possible to perform experiments on a single molecule of DNA to determine the mechanical properties of DNA \cite{5}. An example of such an experiment is the measurement of the forces involved in cellular processes, like transcription of DNA \cite{21}. Yin et al. \cite{21} measured the forces created by RNA polymerases on a DNA strand during the transcription of the DNA to RNA. They were able to get insight into the energy efficiency of this process. Multiple techniques exist to measure the forces that act on DNA, one of the tools commonly used is the optical trap (also known as optical tweezers) \cite{1}. Here a focused laser beam traps a dielectric bead in the center of the laser. A molecule of DNA is attached to the bead. If the bead is pushed out of the center the laser beam is deflected, which causes the bead to be pulled back to the center. By measuring the deflection of the laser-beam the force of the DNA on the bead can be determined. A more detailed description of this mechanism is given in Section 2.1. Various setups of optical tweezers are used in experiments. A single optical trap can be used, in which case one end of the DNA is attached to the bead and the other end is being held by in a fixed point. Other setups use two optical traps, where the DNA is attached to both beads. These setups have the advantage to improve the spatial resolution by exploiting the correlation in the movement of the two beads \cite{13}.

One thing the setup and experiments have in common is the need for a single molecule to be tethered to the bead. This is however not always the case. It is possible for multiple molecules of DNA to attach themselves to the bead. If this happens the process of catching of the DNA must be done again until a single molecule of DNA is caught before the experiment can start. In order to determine if a single or multiple DNA strands have been caught, one can look at the force-distance curve (FD-curve). This means that the beads are pulled apart, and as the distance between them increases, the DNA will start exerting more force on the beads. Since it takes more force to stretch two DNA-molecules than a single molecule a different curve will emerge depending on how many molecules have been caught. This is shown schematically in Figure 1. Currently it is still a user-process to make the determination whether a single molecule has indeed been caught. If this process can be automated, devices, where a bead in an optical trap is used to catch a single DNA molecule for further experiments, become more user-friendly and will be able to used by a people with a wider variety of backgrounds.

Research Goal: Find an automatic way to determine if a single molecule of DNA has been tethered between the beads in the optical trap

1.1 Outline

In this thesis, we will look for a way to differentiate between the multiple tethered possibilities by looking at their respective FD-curves. To come up with a technique for doing this, we will first look at the setup in which these FD-curves are measured. This is done in Section 2, where the experimental setup and the work-flow of a typical experiment is described. Then we will look at the theory behind the force-distance curves and how to automate differentiation between them in Section 3. The implementation of these techniques will be discussed in Section 4.
Figure 1: An qualitative view of the difference in FD-curves for 1 and 2 molecules. The curves are quantitatively described in section 3.

Figure 2: An schematic overview of the optical trap setup being used. Image taken from [9]
2 Experimental setup

The experimental setup that provides the force-distance curves (FD-curves) is a pair of optical tweezers combined with a fluorescence microscope. This combination allows the forces measured by the tweezers to be correlated with the visual output of the microscope.

The experiment takes place on a microfluidics chip, which is divided into 5 separate parallel flow channels. The channels have no physical boundary between them to prevent them from mixing. To make sure the flows do not mix, the flows are kept laminar. In the first channel two polystyrene beads are caught by the tweezers. Then the beads can be moved from channel to channel by moving the chip.

2.1 Optical Tweezers

The optical trap was first described by Ashkin et al. [1], they were able to trap dielectric particles in sizes ranging from 25 nm up to 10 µm. The trapping force is created by the transfer change of momentum of the light-ray when it enters the particles. This is shown in Figure 3

After the beads are trapped, the trap has to be calibrated so that the displacement relative to the trap-center can be converted to a force. The displacement of the bead can be measured by looking at the deflection of the laser. For low forces the bead can be considered to be in harmonic potential. The trapping force is thus be described with a simple spring-model \( F_{\text{trapping}} = -kx \), where \( x \) is the displacement of the particle from the center of the trap and \( k \) is the trap stiffness[15]. The trap stiffness is determined by looking at the Brownian motion of the trapped particle. We follow the derivation described by Berg-Sørensen and Flyvbjerg [2]. We start of by describing the motion of the trapped particle by:

\[
 m \ddot{x}(t) + \gamma_0 \dot{x}(t) + kx(t) = (2k_B T \gamma_0)^{1/2} \eta(t),
\]

(1)

here \( m \) is the mass, \( \gamma_0 \) the friction coefficient and \((2k_B T \gamma_0)^{1/2}\eta(t)\) is the Brownian force. \( \eta(t) \) has the properties:

\[
 \langle \eta(t) \rangle = 0; \langle \eta(t) \eta(t') \rangle = \delta(t - t')
\]

(2)
For the friction coefficient Stoke’s law is used

$$\gamma_0 = 6\pi\mu R,$$

(3)

where $R$ is the radius of the particle, and $\mu$ is the viscosity of the fluid. Since $m \ll \gamma_0$ the term $m\ddot{x}$ can be ignored. This results in (1) becoming:

$$\dot{x}(t) + 2\pi f_c x(t) = (2D)^{1/2}\eta(t)$$

(4)

with $f_c = k/(2\pi\gamma_0)$ and $D = k_bT/\gamma_0$. By measuring the position of the particle for time $T_{\text{max}}$ and taking the Fourier series $\tilde{x}_q = \frac{1}{T_{\text{max}}} \int_{T_{\text{max}}/2}^{T_{\text{max}}/2} x(t) \exp(-2\pi if_q t)dt$ with $f_k = \frac{q}{T_{\text{max}}}$. We can write (4) as:

$$\tilde{x}_q = \frac{(2D)^{1/2}\tilde{\eta}_q}{2\pi(f_c - if_q)},$$

(5)

to remove the complex part we take the absolute value squared:

$$|\tilde{x}_q|^2 = \frac{D}{2\pi^2} \frac{||\eta_q||^2}{f_c^2 + f_q^2}.$$  

(6)

Next we calculate, using our Fourier series definition,

$$\langle \tilde{\eta}_i^* \cdot \tilde{\eta}_j \rangle = \frac{1}{(T_{\text{max}})^2} \int \int \delta(t - t') \exp(2\pi(i\eta_q t - if_j t'))dt'dt$$

$$= \frac{1}{(T_{\text{max}})^2} \int \exp(2\pi i(f_q - f_j) t)dt,$$

(7)

which equals $1/T_{\text{max}}$, if $q = j$, since the term in the exponent becomes 0. This is equivalent to $\langle ||\eta_q||^2 \rangle = 1/T_{\text{max}}$, resulting in

$$P_q = \langle |\tilde{x}_q|^2 T_{\text{max}} \rangle = \frac{D}{2\pi^2} \frac{1}{f_c^2 + f_q^2}.$$  

(8)

In this equation only $k$ is unknown and can thus be determined.

Applying this to our setup yields four different trap stiffness values: $k_{x1}, k_{y1}, k_{x2}$ and $k_{y2}$.

### 2.2 Catching DNA

After the optical traps have been calibrated the beads are moved to the second channel. Here the DNA flows through, after a short while the flow is turned off, so it does not affect the measurements of the force. The beads are now moved close together so that any DNA molecule that has been tethered to one bead can attach itself to the second bead. The beads are then pulled apart and an FD-curve is constructed, by plotting the measured force against the distance between the beads. From this curve the user can determine if a single molecule of DNA has been caught. Note that this is the process we are trying to automate. Instead of the single molecule option it may happen that multiple molecules are tethered or that no molecule has been caught.

Once a single molecule of DNA has been tethered, the beads can be moved to the other channels, where depending on the experiment various proteins and intercalators are flowing. This can be used to insert fluorescent labels into the DNA and interacting protein, which can then be tracked in real time using a scanning laser. For an example, see Heller et al. [9].
3 Theory

3.1 Worm-Like Chain

Various models exist to describe the force-distance curve of DNA. Here we use the worm-like chain model, this is a model which is widely used for fitting force-distance curves [16]. It describes DNA as a continues curve in space \( \mathbf{r}(s) \) with a constant total length of \( l_c \), and it adds a energy penalty for bending of the curve. This energy penalty is dependent on the curvature \( \kappa = \left| \frac{d\hat{t}(s)}{ds} \right| \), where \( \hat{t}(s) \) is the tangent unit vector of the curve defined as \( \partial_s \mathbf{r}(s) \). The energy penalty is \( \int_0^{l_c} \frac{l_p}{2} \kappa^2 ds \), where \( l_p \) is the persistence length, which is the characteristic length at which correlation in the tangent vector die off [12]. If a force \( F \) acts upon the DNA in the z-axis, the total energy of the DNA becomes:

\[
E_{wlc}^{k_bT} = \int_0^{l_c} ds \left( \frac{l_p}{2} \left| \frac{d\hat{t}(s)}{ds} \right|^2 + \frac{F}{k_bT} \hat{z} \cdot \hat{t}(s) \right).
\]  

We define the relative extension as \( r = d/l_c \), where \( d \) is the distance between the two ends. In the case where \( r \approx 1 \), the fluctuations around the z-axis can be considered small if \( l_p \ll l_c \). This is called the semi-flexible approximation which for now we will assume is valid. We now split \( \hat{t}(s) \) in a component \( t_z \) in the z-direction and a component \( \hat{t}_\perp \) perpendicular to the z-axis. By using \( \left| \hat{t}(s) \right| = 1 \) we can write using a Taylor-expansion \( t_z \approx 1 - \frac{1}{2} \left| \hat{t}_\perp \right|^2 / r \). Now we can write \( \left| \frac{d\hat{t}(s)}{ds} \right|^2 \approx \left| \frac{dt}{ds} \right|^2 / r \). This results in (9) transforming to:

\[
E_{wlc}^{k_bT} = \int_0^{l_c} ds \left( \frac{l_p}{2} \left| \frac{dt}{ds} \right|^2 - \frac{F}{2k_bT} \left| \hat{t}_\perp \right|^2 \right) - \frac{F}{k_bT} L. \tag{10}
\]

replacing \( t_\perp(s) \) with a fourier transform \( t_\perp(s) = \frac{1}{2\pi} \int dq \tilde{t}_\perp(q)e^{iqs}, \) where \( t_\perp \) means a single component of the vector. The integral (10) for a single component becomes:

\[
\frac{E_{wlc}}{k_bT} = \frac{1}{2} \int dq \frac{l_p}{2\pi} \left( l_p q^2 - \frac{F}{k_bT} \right) \tilde{t}_\perp^2. \tag{11}
\]

Equipartition of the energy over the modes gives us an expression for \( \langle \tilde{t}_\perp^2 \rangle = \frac{1}{\sqrt{l_p + l_c/k_bT}}, \) This can be used to derive \( \langle |\hat{t}_\perp|^2 \rangle = 2 \int dq \frac{l_p}{2\pi} \langle \tilde{t}_\perp^2 \rangle = \frac{1}{\sqrt{l_p + l_c/k_bT}} \). The 2 before the integral is because there are two components in \( \hat{t}_\perp \). Now we can write

\[
r = \langle t_z \rangle = 1 - \langle |\hat{t}_\perp|^2 \rangle / 2 = 1 - \frac{1}{2\sqrt{l_p + l_c/k_bT}}, \tag{12}
\]

which can be rewritten to

\[
\frac{F}{k_bT} = \frac{1}{4(1 - r)^2}. \tag{13}
\]

This is valid when \( r \approx 1 \). In the case of \( r \approx 0 \) the following relation holds:

\[
\frac{F}{k_bT} = \frac{3}{2r}. \tag{14}
\]

These result where combined by Marko and Siggia into the interpolation formula

\[
\frac{F}{k_bT} = r + \frac{1}{4(1 - r)^2} - 1/4, \tag{15}
\]
which satisfies (13) and (14) in their respective the limits, but in the intermediate region
the solution deviates from the exact solution. In order to correct for this a polynomial
can be added to (15)\[3\], we choose the polynomial suggested by Ogden et al.:

\[
\begin{align*}
F_{oglp} = r + \frac{1}{4(1-r)^2} - \frac{3}{4}r^2. \\
\end{align*}
\](16)

From previous research we know that the persistence length of dsDNA is 50 nm. The
contour-lengths of the DNA-strands we’re looking at are larger than 2 µm, for this reason
we may assume the semi-flexible approximation to be valid.

If there are \(N\) strands tethered to the beads, we make the approximation that the
force will be evenly distributed across all chain. This results in the force per chain being
\(N\) times smaller then the measured force. We now write (15) as:

\[
\begin{align*}
F_{oglp}Nk_B T = r + \frac{1}{4(1-r)^2} - \frac{3}{4}r^2. \\
\end{align*}
\](17)

From this we can infer that if we fit a force extension curve using (16) the resulting
\(lp\) will depend on the number of molecules tethered, if there are more molecules tethered the
persistence length decreases.

3.2 Kernel density estimates

Consider the value of the persistence length \(\tilde{lp}\) retrieved from a fit. There will be a
probability density function (pdf) that describes the chance of finding that value for
all possible tethered situations. If we know the these probability density functions the
probability can be calculated that \(\tilde{lp}\) belongs to the FD-curve of a single molecule. In order
to determine these functions a technique known as kernel density estimation is used. This
will determine the pdf from a series of \(N\) fitted persistence length \(lp_1,\ldots,lp_N\). First define
a kernel function \(K(x)\) with the property

\[
\int_{-\infty}^{\infty} K(x)dx = 1. 
\]

(18)

Now the pdf \(\tilde{f}(lp)\) is defined as:

\[
\tilde{f}(lp) = \frac{1}{Nh} \sum_{i=1}^{N} K \left( \frac{lp - lp_i}{h} \right),
\]

(19)

where \(h\) is known as the bandwidth. This formula can be visualized as adding bumps in
the shape of the kernel function at the measured values, where the bandwidth determines
how wide this bump is. The choice of bandwidth has a large effect on the resulting KDE.
This is shown in fig. 5. If the bandwidth is too small the resulting KDE will be to jagged,
and if the bandwidth is too large the resulting KDE will be too stretched out. How to
choose an appropriate bandwidth will be discussed in section 3.2.2. The choice of the
kernel function it not very important it can be any symmetric probability distribution,
we will use a gaussian distribution here.

The pdf \(\tilde{f}(lp)\) is not constant but will change if the DNA is stretched out further. to
account for this we not only consider the \(lp\) value that was retrieved from a fit but also the
force \(F\) at which the fit was obtained. Thus consider \(N\) data points \((lp_1,F_1),\ldots,(lp_N,F_N)\).
We must now look for \(f(lp|F)\). Let us assume that \(f(lp|F_i) \approx f(lp|F_j)\), if the difference
between \(F_i\) and \(F_j\) is small. We use this to divide the the region \(F = F_{min}\) to \(F = F_{max}\)
into \(M\) bins \(m = 0,\ldots,M-1\). Every fit result gets put in a certain bin based on the force
Figure 4: An example of a KDE, created by drawing 20 samples from a distribution. The kernel functions have been scaled down vertically to improve the clarity of the graph.

associated with it, and over all the values in each bin a KDE is calculated. The index $m$ of the bin as a function of force $F$ is given by

$$m(F) = \lfloor \frac{F - F_{\text{min}}}{F_{\text{max}} - F_{\text{min}}} M \rfloor,$$

(20)

here $\lfloor \rfloor$ means the floor function. The number of data points $N_m$ in the $m$th bin is defined as

$$N_m = \sum_{i=1}^{N} \delta_{m,m(F_i)}.$$

(21)

This can be used to define

$$f(l_p|F) = \frac{1}{N_m(F) h} \sum_{i=1}^{N} K \left( \frac{l_p - l_{p_i}}{h} \right) \delta_{m(F_i),m(F_i)}.$$

(22)

Note that even though persistence length and force were used to illustrate the idea, they can be replaced by any pair of variables.

3.2.1 Classification using KDEs

Let’s say there are $J$ different classes we would like to differentiate between (e.g. a single molecule of DNA, two molecules of DNA). Every class will have its own distribution of persistence lengths associated with it $f_j(l_p|F), j = 1, \ldots, J$. We would like to calculate the probability of a measurement $(l_p, F)$ to belong to class $j$. This can be done by

$$\Pr(j|l_p, F) = \frac{\pi_j f_j(l_p|F)}{\sum_{k=1}^{J} \pi_k f_k(l_p|F)},$$

(23)

where $\pi_j$ is the prior of class $j$ [8]. This is also known as a class weight, it can be used to make the classifier prefer a certain class for instances, when it is known that a certain class is more likely than the others it can be given a higher prior value than the rest. For now we will assume $\pi_j = 1$. (23) will return a probability for each class. Once this probability reaches a threshold value $\Gamma$, we make the classification.
In order to determine how good the classifier works we use a Brier score [4], which is defined as

\[ BS = \frac{1}{n} \sum_{t=1}^{n} (pr_t - a_t)^2. \]  

(24)

Here \( pr_t \) is the probability that an event will happen, \( a_t \) is 1 if the event happened, otherwise it is 0 and \( n \) is the number of classifications made. A Brier score of 0 means perfect accuracy of the classifier, while a score of 1 means that the classifier always fails. Ferri et al. [7] showed that the Brier score can be written as an integral over the threshold \( \Gamma \). They show that for two classes \( j = \{0, 1\} \) and \( n \) test samples, with true positive rate \( F_0(\Gamma) \) (proportion of class 0 points identified as class 0) and the false positive rate \( F_1(\Gamma) \) (proportion of class 1 points identified as class 0)

\[ BS = \int_0^1 [2\pi_0(1 - F_0(\Gamma)) + 2\pi_1(1 - \Gamma)F_1(\Gamma)] d\Gamma, \]  

(25)

where \( \pi_0 \) is \( n_0/n \) and \( \pi_1 \) is \( n_1/n \). Here \( n_0 \) and \( n_1 \) are the number of test samples that belonged to class 0 and class 1 respectively. Now for our case we have more than 2 classes, which we can bring down to two classes by saying that \( j = \{1 \text{ molecule}, \text{not 1 molecule}\} \).

Another difference is that our classifier does not return a static probability but one that changes over time, but this is assumed by Ferri et al. [7]. We can however still use the same mathematical formula to get a score. We will still refer to the quantity calculated by (25) as the Brier score. From (25) we can also extract an optimal threshold by noting that the optimal threshold is probably located at the point where the function inside the integral is minimal.

3.2.2 Automatic bandwidth selection

An important aspect of KDEs is the bandwidth selection, as is shown in fig. 5. For our purpose we need to calculate a KDE for each bin, if the number of bins is very large it is impractical to enter them manually. Multiple techniques have been developed to automate this process. Two of them will be discussed here: Cross-validation and Silvermann’s rule of thumb. First we’ll take a look at cross validation. The idea is comparable to a least-squares method, but the function to minimize in this case is the integrated square estimator defined as

\[ ISE(h) = \int [f(l_p; h) - f_i(l_p)]^2 dl_p, \]  

(26)

where \( f_i(l_p) \) is the actual distribution. This can be rewritten to

\[ ISE(h) = \int f(l_p; h)^2 dl_p - 2 \int [f(l_p; h)f_i(l_p)] dl_p + \int f_i(l_p)^2 dl_p. \]  

(27)

The last integral does not depend on \( h \), so only the first two integrals need to be minimized. The first two terms can be approximated with

\[ CV(h) = \int f(l_p; h)^2 dl_p - \frac{2}{N} \sum_{i=1}^{N} f_{-i}(l_{pi}; h) \]  

(28)

where \( f_{-i}(l_{pi}) \) is the approximation without the \( i \)th data point [19]. By minimizing \( CV \) the optimal bandwidth is retrieved.

Silverman’s rule of thumb is very easy to implement estimation of the bandwidth. The bandwidth is given by

\[ h = \left( \frac{4\sigma_5}{3n} \right)^{1/5}, \]  

(29)

where \( n \) is the number of data points and \( \sigma \) is the standard deviation in the fitted persistence lengths [18].
3.3 Logistic regression

An alternative technique for classification is known as the logistic regression. To explain this technique we will first look at linear regression.

Consider the ith measured result $y^{(i)}$ which depends on $N$ input parameters $x_n^{(i)}$, where $n = 1 \ldots N$. In linear regression we say that:

$$y = h(x),$$

we must now look for a function $h(x)$ which satisfies $y^{(i)} \approx h(x_n^{(i)})$. We choose

$$h(x) = \sum_{n=0}^{N} \theta_n x_n = \theta^T x,$$

which leads us to the problem of finding an appropriate $\theta^T$. Note that there is a term $x_0$ in (31), which is defined as $x_0 = 1$, this is called the intercept term. To solve the problem of finding $\theta$ we define a cost-function $J$ analogues to a least-square optimization

$$J(\theta) = 1/2 \sum_i \left( h_\theta(x^{(i)}) - y^{(i)} \right)^2 = 1/2 \sum_i \left( \theta^T x^{(i)} - y^{(i)} \right)^2,$$

which has a minimum value for the optimal choice of $\theta$.

Linear regression is not a good choice if $y$ is a discrete variable (e.g. $y \in \{0, 1, 2\}$) , which is the case for our problem since $y$ represents the class. First we consider the option where there are only two classes $y = \{-1, 1\}$, now we can define

$$P_+ (y = 1|x) = \sigma(x)$$
$$P_- (y = -1|x) = 1 - \sigma(x).$$

Obviously $\sigma$ should be a function which has a range between 0 and 1. We choose

$$\sigma_\theta(x) = \frac{1}{1 + \exp(-\theta^T x)},$$

This function has three regions: one where it is approximately 0, one where it goes from 0 to 1 (which we will call the transition phase) and a region where the function is approximately 1. This is exactly the behaviour we look for in our classification problems. A plot of this function is shown in Figure 6. Using this definition we can write (33) as $P_\theta (y|x) = \sigma_\theta(yx)$ Analogues to the linear regression we define a cost-function, which is to be minimized

$$J(\theta) = C \sum_i \zeta_\theta(x^{(i)}, y^{(i)}),$$

Figure 5: The effects of selecting the wrong bandwidth

(a) bandwidth too small

(b) bandwidth too large
Figure 6: The $\sigma(z)$ function given in (34) plotted

where $C$ is the penalty parameter and $\zeta$ is the loss-function [6]. The cost function can be derived easily.

We look for $\theta$ which maximizes $\sum_i P_r(y^{(i)}|x^{(i)})$, because a higher $P_r$ will mean a higher probability of $x^{(i)}$ being part of class $y^{(i)}$ (which we know that it is). This is equivalent to maximizing sum of the log-likelihood $\sum_i \log (P_r)$. Now add a minus sign in front to get the function which we should minimize:

$$J(\theta) = -\sum_i \log \sigma_\theta(y^{(i)}x^{(i)}) = \sum_i \log (1 + \exp (-y^{(i)}\theta^T x^{(i)})).$$  \hspace{1cm} (36)

To improve the search for $\theta$ a simple weight decay can be added [11], which transforms (36) into

$$J(\theta) = 1/2 \sum_{n=1}^N [\theta_n^2] + \sum_i \log (1 + \exp (-y^{(i)}\theta^T x^{(i)})).$$  \hspace{1cm} (37)

This has the effect of making the algorithm favour lower values of $\theta$, which means a larger transition phase.

So far we have treated logistic regression in a binary fashion with two classes. In reality we have to take into account the existence of more classes, this can be done by using a “one-vs-rest” approach.
4 Implementation

The program is designed in such a way that it can run as a separate program where it communicates with the software that controls the experiment. The communication is done using system pipes. A description of the protocol is given in appendix A.1.

4.1 Curve generation

Ideally all the curves are from actual measured data. There was no data available to us of FD-curves for the no molecule and the 2 molecules case. To generate curves for these cases (4) was solved numerically using a technique described by Volpe and Volpe. It using the following equation for calculating the positions at i-th simulation step:

\[ x_i = x_{i-1} - (k_x x_{i-1} - F_x) \Delta t / \gamma + \sqrt{2D \Delta t} \eta(t), \]

(38)

here \( x_i \) is the x position at the i-th step and \( \Delta t \) is step size. \( \eta(t) \) is generated with the matlab command \( \text{randn()} \). The equation for the y and z coordinate are analogues to (38).

This technique can easily be used to simulate FD-curves. We do this by looking for a function which describes the force that acts on the bead as function of distance. The worm-like chain model could be used for this case, however we would like to include experimental defects in our simulation which makes the wlc-curve not ideal. In order to get a simulated FD-curve we calculate \( \langle F_{\text{measured}} \rangle (r) \) and \( \sigma_{\langle F_{\text{measured}} \rangle}(r) \) for known curves of 1 molecule. This is done by using the KDE result for 1 curve shown in Figure 7. We now define the FD-curve for n molecules used in the simulations as \( F_{\text{sim}}(r|n) = n \langle F_{\text{measured}} \rangle (r) + \alpha(r) \sigma_{\langle F_{\text{measured}} \rangle}(r) \), where \( \alpha(r) \) is a straight line going through the points \((r_1, \nu_1)\) and \((r_2, \nu_2)\). \( \nu \) is a random number between -2 and 2, and \( r_1 \) and \( r_2 \) are the begin and end points of the simulation.

4.2 Curve fitting

To get the persistence length and contour length from a measured FD-curve according to (16) we use a least-squares optimization. We refer to (16) as \( F(\nu|l_c, l_p) \). If there are N data points measured \( i = 1 \ldots N \), we say that for the optimal values of \( l_p \) and \( l_c \) the sum

Figure 7: The kernel density estimate for 1 molecule case, this is used as bases to simulate curves for multiple molecules.
Figure 8: The $S$ landscape for $l_p = 52$ nm and $F_{\text{max}} = 5$ pN. We can clearly see if we fit with (16) we may not find a global minimum, but if we fit (40) a global minimum is always found.

of the squared residuals $S$ must be minimal:

$$S(l_c, l_p) = \sum_{i=1}^{N} [F_i - F(z_i|l_c, l_p)]^2,$$

where $F_i$ and $z_i$ are the force and the distance measured respectively at the $i$th data point.

We set the following bounds for our problem:

$$0 \leq l_p \leq l_{p_{\text{max}}},$$
$$0 \leq l_c \leq l_{c_{\text{max}}},$$

where $l_{c_{\text{max}}}$ is the maximum contour-length we should consider in the fitting procedure, and $l_{p_{\text{max}}}$ the maximum value for the persistence length. The value of $l_{p_{\text{max}}}$ we choose to be 300 nm. The value of $l_{c_{\text{max}}}$ can be set by the user, but the default value is 20 µm, which is larger than normal contour-lengths used in the experiments.

If we look at the landscape of $S$ for a fit at various forces, we see that if the force increases the minimum in the landscape becomes more pronounced, but when the forces get too high the minimum shifts to smaller values of the persistence length. This is shown in fig. 9. If we look at a slice of the landscape of $S$, we see a problem when just using (16). This is shown in fig. 8a. If we start off with an initial guess of the contour length which is too low, we get stuck in local minima instead of finding a global minimum. The erratic behaviour of the $S$-landscape for lower contour lengths can be explained by noting that $z > l_c$ (16) moves down towards 0 pN again. To correct for this behaviour we use the following FD-equation for fitting

$$F^*(z|l_c, l_p) = \begin{cases} 
F(z|l_c, l_p) & \text{if } F(z|l_c, l_p) < 65 \text{ pN} \\
65 \text{ pN} & \text{if } z > l_c \\
65 \text{ pN} & \text{otherwise}
\end{cases} \quad (40)$$

This eliminates the erratic behaviour at lower contour lengths as shown in fig. 8b. However the contour-length of the DNA is known in advance, so it can be set constant when fitting the data.

4.3 Classification

To make the classification described in section 3.2 and section 3.3, we use the 'scikit-learn' package for python [17]. It includes KDE-learning and logistic regression. To be able to classify unknown curves we must first do some preparation:
First there should be a database of curves for which we know the classification. This is then subdivided into two sets: the training-set and the test-set.

The analysis modules are executed on each of the curves in the training set. KDE-learning and logistic regression are performed on the results of the analysis modules. The results are stored in a KDE-database.

Looking up the probabilities of a fit for an unknown curve is done by using the KDE-database and (23), this will return a probability for each of the classes. It is possible to use the probabilities from multiple KDE-databases by averaging over them. If the resulting probability exceeds a threshold $\Gamma$ for a class it will be classified as belonging class. Logistic regression can then used as an additional verification by checking if it yield the same classification.

The test-set is used to check the performance of the classification. By looking at the resulting classification for a certain threshold, the false positives per class (classified as 1 molecule, but it was not) and false negative percentage (it was 1 molecule, but failed to classify as such) can be calculated and plotted. The graph where the false positives and false negatives are plotted against the threshold will be referred to as the "performance graph".

The database we used contained 50 curves for the no molecule case, 50 curves for the 1 molecule case and 50 curves for the 2 molecule case, but only 4 in the 1 molecule case were actual measured curves. The rest was obtained using simulations. We used 50% for training and 50% for testing. All our curves are from pkYb1 DNA, which has a contour-length of 2.853 $\mu$m.

**Figure 9:** A couple of residuals landscapes for various maximum forces. This shows that the fitted values become more accurate at larger forces, but if the forces get too large the persistence length starts to be underestimated as predicted [3]. The actual contour length is 2.83 $\mu$m and the persistence length is 56 nm.
5 Results

The created KDEs are shown in Figure 10 and Figure 11. The resulting performance graphs are shown in Figure 12. This shows that the \((l_p,F)\) approach performs worse than the \((F,r)\), especially in the classification of the no molecule case. We explain this by noting we have observed that the fitted persistence length of the no molecule case starts between 25nm and 50nm, and only after 90% extension is reached the persistence length makes a rapid transition towards the maximum value of 300 nm. So when the classification starts the fitted persistence length is still in the 1 molecule region for the no molecule curve, this causes the misclassification. Later on we will look at the performance when the classification starts at 95% extension.

The increase in the false negatives at higher threshold is expected and can be explained by noting that threshold is too high: The probability never exceeds the threshold and thus no classification takes place. If the threshold is too low the classification is not reliable enough which creates both false positives and false negatives. Both these behaviors manifest themselves in the performance graphs. In an actual experiment the performance graphs could be used to set an acceptable threshold. In the most optimal scenario there will be no false negatives and no false positives, we can see that this is not the case. The user can look at the performance graph and determine what an acceptable percentage of false positives is. The false negatives is not as important since it does not "contaminate" the end results of the experiment, but if the value is too high the experiment would take longer. So it would be a choice depending on the experiment that is being performed. The Brier score works pretty well to attach a single number to these graphs, the graphs that look better have a lower Brier score as is expected. The Brier score and the optimal threshold are not perfect yet, since it does not take into account that false negatives are preferred over false positives (we would rather have fewer results than wrong results). The optimal threshold is also not a bad approximation for what would be a good value for an experiment.

For the \((l_p,F)\) one of the striking features is the bump in the false positives for the 2 molecule case. This can be explained by looking at the probabilities of the 2 molecules curves over time. At \(t = 0\) the probability for 2 molecule case will exceed the 1 molecule case, however as time progresses the probability for the 1 molecule case increases and overtakes the probability of the 2 molecule case while the 2 molecule case probability remains constant. After a little while longer the probability for the 2 molecule case starts to increase and eventually it will overtake the 1 molecule case. To summarize:

- Lower threshold: quicker classification, the 2 molecule case has a higher probability, resulting in correct classification
- Intermediate threshold: longer wait before classification, the probability in the 1 molecule increase has overtaken the 2 molecule case. Resulting in incorrect classification
- High threshold: long wait before classification, the probability of the 2 molecule case eventually overtakes the 1 molecule case. Resulting in correct classification.

One could potentially solve this problem by waiting until a certain relative extension is reached before making the classification. We tried this by waiting until 95% extension was reached, which could also solve the no molecule classification problem discussed earlier. The results are shown in Figure 13. They do show an improvement compared to the original result, it is however not better than the \((F,r)\) approach. The approach where the 2 resulting probabilities are averaged (Figure 12c) did not show any significant improvement over the raw data. This all makes us suspect that the best approach is simply using the raw FD-data.

Note that we used mainly simulated curves instead of actual measurements, the validity of this will be discussed in Section 6. Additionally we checked how well a pure logistic
Figure 10: The KDEs created for the 1 molecule case, the no molecule case and the 2 molecules case.
A prior was used to set the maximum of the probability density functions to 1 in order to improve the clarity of the images.

classifier works, the performance graph is shown in Figure 14. This shows that just using the logistic regression is does not have better performance than the kernel density estimation. Note that for the other performance graphs logistic regression was not used, since for the \((F, r)\) case the curve can’t be described by the simple logistic regression, that would require adding higher order terms to the sigmoid function instead of using just linear terms. Since the classes can not be separated using straight lines in the \((r,F)\) plane.
Figure 11: The KDEs created for the 1 molecule case, the no molecule case and the 2 molecules case. A prior was used to set the maximum of the probability density functions to 1 in order to improve the clarity of the images.

Figure 12: The performance graph of the various different KDEs
**Figure 13:** The performance graph of the \((l_p, F)\) approach where the decision is only made after 95% extension

**Figure 14:** The performance graph of the \((l_p, F)\) approach where only logistic regression was used
6 Outlook and Conclusion

One of the first things that should be done is to create a database featuring actual measured curves instead of simulated ones. We expect the simulated results to be quite accurate for the no molecule case, but for the 1 and 2 molecule case we do not rely on physics for the FD-curves but on statistics. This may work for the 1 molecule case where we have some known curves. For the 2 molecule case however we make the assumption that the 2 molecules do no interact with each other. However we expect some interactions between them which would have an effect on the FD-curve. In our simulations we also do not take into account the binding positions of the molecules on the bead, we assume the molecules are attached at the points on the bead, where the distance between the bead is the shortest. In reality the 2 molecules are probably more likely to be attached to different parts of the bead. This would mean that the 2 molecules would not be attached to the points of shortest distance. That will be beneficial to the classification algorithm, because we assume the shortest distance is the molecule extension, and if the actual extension of the molecules is larger than the shortest distance a higher force is measured. If there is a higher force being measured for the 2 molecules than is simulated the actual FD-curves would deviate more from the 1 molecule case than is being simulated, which would make for easier classification. Note that for the pkYB1 DNA with a contour-length of 2.853 µm and a bead diameter of 1 µm, this effect could be detectable. The only way to be sure is to have actual 2 molecule FD-curves.

The kernel density estimation using $F$ and $r$ yielded better results than the $F$ and $l_p$ combination, even if we added additional constraints the $(l_p,F)$ approach specifically chosen to improve the classification. This makes us suspect that using the raw FD-data is better than fitting. An additional advantage of the $(F,r)$ approach is that it can be used in a wider range of system, where there is an FD-curve but no theoretical description exists.

One could improve the technique we describe here by adding "memory", which means that instead looking at the current measured point for classification it would take into account the previous data.

The results here were derived from using 1 combination of training/test curves. In order to validate the technique multiple combination of training/test curves should be used. If the classification technique is valid and the database large enough, we expect there to be no virtually no difference between the various permutations.

We used the Brier score here to determine how good the classifier performed, this worked pretty well for a first attempt, since the scores where intuitively understandable and the optimum threshold was never far from a realistic value. It should however be modified to take into account that we rather misclassify a 1 molecule as something else than vice versa. One can also look how far along the curve we have to go before the classification is made. Consider two classifiers that perform equally well but one makes the classification much quicker, it is probably a better choice.

In conclusion we have tried to classify FD-curves using kernel density estimations. We looked at the differences between the naive approach of simply using the raw FD-data compared to the more complex approach of fitting a physical model to the curve and using the result for classification. The performance of the fitting approach was not better than the naive approach. The result of our measurement should be validated first using multiple training/test sets and real data. If the results remain the same, we suggest that it is better to focus on improving the naive approach by using for example a neural network with a memory. This has the added advantage of being applicable to a wider range of physical systems than just fitting the worm-like chain model.
References


A Protocol Description

A.1 IPC protocol

The software for data analysis/analysis module is referred to as A. the software that provides the data is referred to as B.

Create a connection

- B creates two pipes: one for sending and one for receiving e.g. "pipeWrite" for sending and "pipeRead" for receiving
- A connects to the two pipes
- A should use "pipeWrite" for reading and "pipeRead" for writing

Both A and B can write commands in their respective pipe. A command is a single line of text followed by a newline character (\n)

Handshake

- B writes: identify.
- A returns: identify -name NAME_OF_PROGRAM -version VERSION_NUMBER.
- B writes: get -output.
- A returns the output. every output parameter is a line:
  output -float VARIABLE_NAME.
Example for a worm-like chain fitter with two return parameter lc and lp:
1. output -float lc.
2. output -float lp.
- After writing the output A writes the input values as: input -float VARIABLE_NAME DEFAULT_VALUE.
Example for a worm-like chain fitter which requires the contour length to be known:
input -float lc 2.8.
This means that A by default assumes the contour-length to be 2.8 micrometer

Note: The units that should be used are given at the end of this document.

Starting the experiment

- Before starting the experiment values of the input of A can be set by B by writing:
  input VARIABLE_NAME VALUE. If this is not done A will assume the default values set in the handshake.
Example for lambda-dna in a worm-like chain fitter which requires the contour length:
input lc 16.5
- B writes: start DATA_FREQUENCY BEAD_DIAMETER.

Sending and receiving experimental data

- B sends data by writing: data TIME_STAMP FORCE_1 FORCE_2 DISTANCE
- A returns results by: return_data PARAMETER_1 ... PARAMETER_n TIME_STAMP
In the WLC-fit example, if lc would be 2500 nm and lp 50 nm after 1410ms, A would write:
return_data 2.500 0.050 1410 The order in which the parameters are written is the
same as the order in which they were send with the "output" command during the handshake.

Note: not for every data point send by B, does A have to return a return_data command.

Stopping the experiment

- B writes: stop

After the experiment has been stopped it can be started again using the "start" command

Disconnecting

Any of the two program can just close one of the two pipes to disconnect.

Units

To have a constant experience between multiple data analysis modules. The following units should be used:

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length</td>
<td>micrometer</td>
</tr>
<tr>
<td>Force</td>
<td>pN</td>
</tr>
<tr>
<td>Time</td>
<td>ms</td>
</tr>
<tr>
<td>Frequency</td>
<td>hz</td>
</tr>
</tbody>
</table>

B Code listing

B.1 Analysis Module

Listing 1: receiver.py

```python
# coding: utf-8

---

This file contains a class to handle the input from the experiment
Created on Fri Apr 14 12:01:07 2017

@author: s115898

---

import time
import numpy as np
import pipes as p
import data_threader as dt
import configparser as config
import sys as s
s.path.insert(0, '../Utils/)
import kde_database as database

class Receiver:
    
    ***class to handle the data coming from the experiment***
    BUFFER_SIZE=4

    def __init__(self, thread):
        
        ***initializes the receiver:
        thread: a reference to a class of type DataThread
        ***
```
self.pipe_read = p.create_pipe(True)
self.pipe_write = p.create_pipe(True)
self.thread_type = thread
self.connected = False

# information on the output nodes
self.output_list = []

# read settings.ini to get the info settings
cfg = config.ConfigParser()
cfg.read("settings.ini")
self.training_mode = cfg.getboolean("info","training mode")

# load the kde database if necessary
self.kde = []
self.calibration = None

if self.training_mode is False:

    # getting the kde files
    self.kde_database_files = cfg.get(thread.NAME, "KDE database")
    self.kde_database_file_list = self.kde_database_files.split(';

    # retrieving the weight values for multiple kdes
    weights = np.ones(len(self.kde_database_file_list))
    if cfg.has_option(thread.NAME,"KDE weights"):
        weight_string = cfg.get(thread.NAME,"KDE weights")
        weight_list = weight_string.split(';
        for i in range(0,len(weight_list)):
            weights[i] = float(weight_list[i])

    # loading the kdes
    index = 0
    for file in self.kde_database_file_list:
        print("loading kde database: " + file)
        kde_to_add = database.KDEDatabase(file)
        kde_to_add.weight = weights[index]
        self.kde += [kde_to_add]
        index += 1

    # load the calibration
    if cfg.has_option(thread.NAME,"Calibration"):
        calibration_file = cfg.get(thread.NAME,"Calibration")
        print("loading calibration: " + calibration_file)
        self.calibration = database.Calibration(calibration_file)

def run(self):
    """contains the actual program loop""
    self.connect_pipes()
    # handle the input while the pipes are connected
    while self.connected:
        # create a thread for the analysis
        self.thread = self.thread_type()
        # set whether in training mode
        self.thread.training_mode = self.training_mode
        self.thread.kde = self.kde
        self.thread.calibration = self.calibration
        self.thread.output_list = self.output_list
        # send the thread the pipe which output is written
        self.thread.set_pipe(self.pipe_write)
        # perform the setup

    def run(self):
self.thread.setup()

#handshake

self.handshake_and_wait_for_start()

#data is being received

if self.connected:
    self.read_data_and_wait_for_stop()

#stop the thread

self.thread.stop()

del self.thread

print("connection lost")


def connect_pipes(self):
    """this function runs until both pipe are connected""
    pipe_open_try_count = 0  #counts the number of connection
    while self.pipe_read.open_pipe("\\\\.\\pipe\\lumicks") is False:
        print("could not open pipe")
        pipe_open_try_count += 1
        if pipe_open_try_count == 50:
            return
        time.sleep(1)  #wait a second befor retrying

    pipe_open_try_count = 0
    while self.pipe_write.open_pipe("\\\\.\\pipe\\lumicks_return")
        is False:
        print("could not open return pipe")
        pipe_open_try_count += 1
        if pipe_open_try_count == 50:
            return
        time.sleep(1)

    self.connected = True


def handshake_and_wait_for_start(self):
    """this function performs the handshake and sets everything up for the experiment""

    while True:
        message = self.pipe_read.read()
        message_split = message.split(" ")
        if len(message) == 0:
            #pipe connection has been lost
            self.connected = False
            return
        if message == "identify\n"
            self.pipe_write.write("identify -name "+self.thread.NAME
            
            " -version "+
            self.thread.VERSION+"\n")
        elif message == "get -output\n"
            self.thread.init_output()
            self.output_list = self.thread.output_list
            if self.training_mode is False:
                """write custom outputs when returning probabilities""
                for kde_groups in self.kde[0].get_ordering():
self.thread.pipe.write("output -float "+
  kde_groups.replace(" ", "")+"\n")

elif message_split[0] == "input":
    self.thread.on_input(message_split[1], float(
        message_split[2]))

elif message_split[0] == "start":
    # the experiment is starting
    self.bead_size = float(message_split[1])
    self.data_frequency = float(message_split[2])
    # initialize the buffer with a size of 4 seconds and 4 columns
    self.data_buffer_width = self.thread.input_count
    self.data_buffer = \n    np.zeros([int(self.data_frequency*self.BUFFER_SIZE),
               self.data_buffer_width])
    self.data_buffer_size = int(self.data_frequency*self.
        BUFFER_SIZE)
    self.data_buffer_index = 0 # start at beginning
    self.data_buffer_filled = False # buffer is not yet fully filled
    self.thread.on_start()
    return

def read_data_and_wait_for_stop(self):
    """this function reads the incoming data from the pipe and
    writes it to
    the buffer until the experiment is stopped""
    # start the data analysis thread
    self.thread.start()
    # give the thread some info on the buffer dimension and location
    self.thread.set_buffer_info(self.data_buffer_size, self.
        data_buffer_width, self.data_buffer.data)
    # while data is being received
    while True:
        message_full = self.pipe_read.read()
        if len(message_full) == 0:
            # connection lost
            self.connected = False
            break
        # split the message into lines
        message_lines = message_full.split("\n")
        # for every received line, check what the command is
        for message in message_lines:
            message_split = message.split(" ")
            if message_split[0] == "data":
                # data has been received
                if len(message_split) < 5:
                    break
                # time_stamp=float(message_split[1])
                f1=float(message_split[3])
                f2=float(message_split[4])
                d=float(message_split[2]) - self.bead_size
                """Only send data for forces below MAX_FORCE to the buffer
                otherwise keep the buffer empty until the forces dip below"""
MAX_FORCE again

```python
if f2<self.thread.MAX_FORCE or f1<self.thread.MAX_FORCE:
    # sends data to the buffer
    for index in range(0, self.data_buffer_width):
        self.data_buffer[self.data_buffer_index] = float(message_split[index + 1])
    # subtract the bead size from the distance
    self.data_buffer[self.data_buffer_index, 1] = self.bead_size
    # notify the data analysis thread
    self.thread.set_buffer_index(self.data_buffer_index, self.data_buffer_filled)
    # increase the write index
    self.data_buffer_index += 1
    # check if end of buffer is reached
    if self.data_buffer_index == self.data_buffer_size:
        # start from the beginning
        self.data_buffer_index = 0
        # the buffer is now fully filled
        self.data_buffer_filled = True
    else:
        # clears the buffer
        self.data_buffer_index = 0
        self.data_buffer_filled = False
        # notify the data analysis thread
        self.thread.set_buffer_index(0, False)

elif message_split[0] == "stop":
    # stop the experiment
    self.thread.stop()
```

Listing 2: data_threader.py

```bash
# coding: utf-8
#
# @package DataAnalysis
#
# This file contains a thread class to do the actual data analysis.
In order to implement a data analysis routing please extend the
DataThread class
and implement the init_output and the perform_output function
Created on Fri Apr 7 15:11:13 2017

@author: s115898
```
import numpy as np
import pipes as p
import sys as s
s.path.insert(0, '../Utils/)
import kde_database as database
import data_averaging as average

# Documentation for a class.
# More details.
class DataThread(threading.Thread):
    """A thread which run the data analysis part of the software
    """
    NAME="data thread"
    VERSION="0.1"
    MAX_FORCE=50
    # Minimum buffer size before perform_test is called
    MINIMUM_BUFFER_SIZE=200
    # Column which stores time
    TIME = 0
    # Column which stores distance
    DISTANCE = 1
    # Column which stores force on bead 1
    FORCE1 = 2
    # Column which stores force on bead 2
    FORCE2 = 3
    """FORCE1_X = 4
    FORCE1_Y = 5
    FORCE2_X = 6
    FORCE2_Y = 7
    BEAD1_X = 8
    BEAD1_Y = 9
    BEAD2_X = 10
    BEAD2_Y = 11"
    ALLOW_DECISION = True
    ENABLE_AVERAGING = False
    AVERAGE_LENGTH = None
    AVERAGE_OUTPUT = None

    def __init__(self):
        threading.Thread.__init__(self)
        # Boolean to indicate if the thread should keep running
        self._can_run = True
        # The size of the buffer
        self._buffer_size = 0
        # The index at which data is written in the buffer
        self._buffer_index = 0
        # The number of columns in the buffer array
        self._buffer_width = 0
        # The buffer array
        self._buffer = 0
        # Boolean to indicate whether the buffer is filled
        self._buffer_filled = False
        # Numpy array that holds the data which is to be analyzed
self.data = np.zeros(1)
#the pipe in which to return the data
self.pipe = p.create_pipe()
##the last time stamp which a data analysis test was performed
self.last_time_stamp = -1
##the number of columns in the database
self.input_count = 12
##list of output variables
self.output_list = []
##mode information
self.training_mode = None
## contains the kde database initialized by the receiver class
self.kde = None
## The result average, initialized by the receiver class
self.averager = None
## option calibrator for output probabilities, initialized by
# the receiver class
self.calibration = None
##additional mode information
self.output_in_test_mode = False

def run(self):
    print("running thread")
    if self.ENABLE_AVERAGING:
        self.averager = average.Averager(self.get_output_count(),
                                            self.AVERAGE_LENGTH)
    while self._can_run:
        #first copy the buffer in data_to_analyze
        p1, p2, p3, p4 = self.get_data_buffer_positions()
        if p4 != p1: #only false when p1==p4==0.
            self.data = np.vstack((self.buffer[p1:p2][:],
                                    self.buffer[p3:p4][:]))
        #test if the buffer has enough data in it to perform an
        analysis
        if self.data.size > self.MINIMUM_BUFFER_SIZE*self.buffer_width:
            if self.last_time_stamp != self.data[-1, 0]:
                self.perform_test()
            self.last_time_stamp = self.data[-1, 0]
            print("finish thread")

    ##sets the pipe to which the results should be written
    # @param return_pipe the pipe
    def set_pipe(self, return_pipe):
        self.pipe = return_pipe

    ##Stops the thread
    def stop(self):
        self._can_run = False
        self.on_finish()

    ##Set information about the buffer
    # @param size the size of the buffer
    # @param width the number of columns in the buffer
    # @param pointer pointer to the memory of the buffer
    def set_buffer_info(self, size, width, pointer):
self.buffer_size = size
self.buffer_width = width
self.buffer = np.frombuffer(pointer).reshape((size, width))

## called when the analysis program should define its outputs/inputs
def init_output(self):
    pass

## replaces the init function, to be implemented by the child class
def setup(self):
    pass

## called when the experiment sets an input variable
# @param name the name of the input
# @value the value of the input
def on_input(self, name, value):
    pass

## called when the experiment is stopped
def on_finish(self):
    pass

## Returns the part of the buffer that was not available in the last perform_test_call
# @return a tuple (a,b) where the new data is stored in range(a,b)
def get_new_data_range(self):
    if self.last_time_stamp < 0:
        return (0, self.data[:, 0].size)
    previous_index = np.where(self.data[:, 0] == self.last_time_stamp)[0][0]
    max_index = self.data[:, 0].size
    if previous_index == (max_index - 1):
        return (previous_index, previous_index)
    return (previous_index + 1, max_index)

## Called when the experiment is started
def on_start(self):
    pass

## Creates an output in the experiment software
# @param name Name of the output (cannot contain whitespaces)
def create_output(self, name):
    self.output_list += [name]
    self.pipe.write("output -float " + name + ":f\n")

## Creates an output in the experiment software
# @param name Name of the input (cannot contain whitespaces)
# @param default_value default_value of the input
def create_input(self, name, default_value=30):
    self.pipe.write("input -float " + name + " {f}\n".format(" + default_value))

## writes the output of the program to the experiment
# @param **kwargs kwargs in the form outputname=value
def write_output(self, **kwargs):

time_stamp = "{:f}".format(self.data[-1, 0])
if self.training_mode is True:
data_string = ""
    for output in self.output_list:
        data_string += "{:f} ".format(kwargs[output])
    self.pipe.write("return_data "+data_string+time_stamp+"\n"

    )
else if self.ALLOW_DECISION:
    xvalue = args[self.kde_xvariable_index]
yvalue = args[self.kde_yvariable_index]
    probabilities = self.kde[0].weight*self.kde[0].
        get_kde_probabilities(**kwargs)
    ordering = self.kde[0].get_ordering()
    total_weight = self.kde[0].weight
    for kdes in self.kde[1:]:
        probabilities += kdes.weight*kdes.
            get_kde_probabilities(ordering, **kwargs)
        total_weight += kdes.weight
    probabilities = probabilities/(total_weight)
    if self.ENABLE_AVERAGING:
        probabilities = self.averager.add_value(kwargs[self.
            AVERAGE_OUTPUT],probabilities)
    if probabilities is not None:
        data_string = ""
        for output in self.output_list:
            data_string += "{:f} ".format(kwargs[output])
        #use the calibration to improve probabilities
        if self.calibration is not None:
            probabilities = self.calibration.probabilities( 
                ordering, probabilities,**kwargs)
        self.pipe.write("return_data "+data_string+"{:f} ". 
            len(probabilities)).format(*probabilities)+ 
                time_stamp+"\n")
## sets the buffer index at which the data is currently being 
written
    # this index is used to determine the data in the buffer which can 
be
    # analysed.
    # @param index index of the buffer where data is being written
    # @param filled whether the buffer is full
    def set_buffer_index(self, index, filled):
        self.buffer_index = index
        self.buffer_filled = filled

    def get_data_buffer_positions(self):
        """returns a tuple (p1,p2,p3,p4) that indicates which part of 
the buffer can be read safely for analysis. The buffer part that can be 
read safely 
is self.buffer[:p1] and self.buffer[p3:p4]"
        if self.buffer_filled is False:
            return (0, 0, 0, self.buffer_index)
        else:
            return (self.buffer_index+1, self.buffer_size-1, 0, self.
                _buffer_index)
# implemented by child to perform the actual test. the data that
# can be tested is stored in self.data. To return the result of this
# function use the write_output function
def perform_test(self):
    pass

## returns the number of output variables
# @return the number of output variables
def get_output_count(self):
    if self.training_mode:
        return len(self.output_list)
    else:
        return len(self.kde[0].get_ordering())

# This is a minimal example of how to use DataThread

Listing 3: pipes.py

# coding: utf-8
***
This file contains classes to work with named pipes
use the function create_pipe(). It will return a pipe based on the
current platform.
Created on Thu Apr 13 11:23:39 2017
@author: s115898
***
import sys
class NamedPipe:
    """the base class for working with pipes, needs to implemented to
work on specific platforms""
def __init__(self):
    self.is_open=False
def open_pipe(self,name):
    """opens a pipe with name, implementations should return true
on succes
otherwise return false""
pass
def read(self):
    """returns the text in the pipe, implementation should return
an empty string
if the pipe is closed. otherwise they should wait until
something can be read""
pass
def write(self,string):
    """writes string to the pipe"
pass
***
a default QPipe class which is used to implement pipes on non windows platform.

Note: this has not yet been tested. the implementation has been tested and works on windows. Although the implementation seems to be slightly less reliable than the native windows version (needs more testing)

```python
from PyQt5.QtNetwork import QLocalSocket
from PyQt5.QtCore import QByteArray

# a cross platform implementation using PyQt5

class QPipe(NamedPipe):
    def __init__(self):
        self.is_open=False
        self.socket=QLocalSocket()
        self.lines_to_read=[]

    def open_pipe(self,name):
        self.socket.connectToServer(name)
        if self.socket.waitForConnected(100) == True:
            self.is_open = True
            return True
        return False

    def write(self,string):
        if self.is_open:
            self.block = QByteArray()
            self.block.append(string)
            self.socket.write(self.block)
            self.socket.waitForBytesWritten(-1)
            self.socket.flush()
            return True
        else:
            return False

    def read(self):
        if len(self.lines_to_read)>0:
            #print("multiline error")
            #print(self.lines_to_read)
            a=self.lines_to_read[0]
            del self.lines_to_read[0]
            #print("sending: "+a)
            return a+'\n'
        while self.socket.waitForReadyRead(100) == False:
            # check if the socket is closed
            if self.socket.state() == QLocalSocket.UnconnectedState:
                return ""
            lines= self.socket.read(4096).decode("utf-8").split("\n")
            # more than 1 line was read
            if len(lines)>1:
                self.lines_to_read+=lines[1:-1]
            if len(lines[0])> 0:
                return lines[0]+'\n'
            else:
                return "void\n"
        # just one line was read
```
if len(lines[0]) > 0:
    return lines[0] + '
' else:
    return "void
"

## implement a windows only version of named pipes, if the windows
# is used
if sys.platform.startswith("win"):
    import win32file

class WinPipe(NamedPipe):
    """ a windows only implementation of the pipes""

def __init__(self):
    self.is_open = False

def open_pipe(self, name):
    try:
        self.fileHandle = win32file.CreateFile(name,
            win32file.GENERIC_READ |
            win32file.GENERIC_WRITE,
            0, None,
            win32file.OPEN_EXISTING,
            0, None)
        return True
    except win32file.error:
        return False

def read(self):
    try:
        data = win32file.ReadFile(self.fileHandle, 4096)
        return (data[1]).decode("utf-8")
    except win32file.error:
        return ''

def write(self, string):
    try:
        win32file.WriteFile(self.fileHandle, bytes(string, 'utf
' -8'))
        win32file.FlushFileBuffers(self.fileHandle)
        return True
    except win32file.error:
        return False

## create the actual pipe
# @param force_qt boolean to indicate whether a qt pipe should always
# be used

def create_pipe(force_qt = True):
    """ this function should be called to create a pipe
    creates a WinPipe on windows unless force_qt is true.
    On all other platform it creates a QPipe""
    if sys.platform.startswith("win") and not force_qt:
        return WinPipe()
    else:
Listing 4: data_averaging.py

```python
import numpy as np
class Averager:
    def __init__(self, input_size, average_distance):
        self.total = np.zeros(input_size)
        self.size = 0
        self.input_list = []
        self.value_list = []
        self.average_distance = average_distance

    def add_value(self, value, input_array):
        self.total += input_array
        self.size += 1
        if self.average_distance > 0:
            self.input_list += [input_array]
            self.value_list += [value]
            if value > self.value_list[0] + self.average_distance:
                del self.value_list[0]
                self.size -= 1
                self.total -= self.input_list[0]
                del self.input_list[0]
        return self.total / self.size
```

Listing 5: wormlikechain_fitting_lock_contour_length.py

```python
import numpy as np
import scipy.optimize as optimization
import wormlikechain_numerical as w
import data_threader as data
import receiver as r
```

A data analysis program that tries to perform a WLC-fit on the data
With an additional feature of locking the contour length
def wlc_interpolation_fit(extension, force, lc, guess=(50),
    minimum=(0), maximum=(300)):
    """performs a non linear least squares using a wlc interpolation
    formula. returns a tuple of (l,p) or None if the fit failed""
    def wlc_interpolation(extension, lp):
        return value = w.lc_interpolation(extension, lc, lp)
        return value[extension > lc*0.97] = 65
    try:
        param, cov = optimization.curve_fit(wlc_interpolation,
            extension, force,
            np.array([guess[0]]),
            method='trf', bounds=(minimum, maximum)
        )
        return param
    except RuntimeError:
        return None

class WLCFit(data.DataThread):
    """this is the thread that does all the actual calculations""
    NAME = "wlc_fit_lock_lc"
    VERSION = "0.2"

    def setup(self):
        self.MAX_FORCE = 20
        self.guess = [50]
        self.contour_length = 16500
        #set up averaging
        self.ENABLE_AVERAGING = True
        self.AVERAGE_LENGTH = 0.05
        self.AVERAGE_OUTPUT = 'z'
        self.output_in_test_mode = False

    def init_output(self):
        """tells the experiment what data is being sent back""
        self.output_in_test_mode = False
        self.create_output("l_p")
        self.create_output("force")
        self.create_output("z")
        self.create_input("force_limit", 20)
        self.create_input("l_c", 16.5)

    def on_input(self, name, value):
        if name == "force_limit":
            self.MAX_FORCE = value
        if name == "l_c":
            self.contour_length = value*1000

    def perform_test(self):
parameters = wlc_interpolation_fit(self.data[:, self.DISTANCE] * 1000, 
self.data[:, self.FORCE2], 
self.contour_length, 
self.guess)

if parameters is not None:
    self.guess = parameters
Faverage = np.average(self.data[self.data[:, self.FORCE2].size - 40:, self.FORCE2])
rel_extension = self.data[-1, self.DISTANCE] / (self.contour_length / 1000)

if Faverage > 5 or rel_extension > 0.95:
    self.ALLOW_DECISION = True
else:
    self.ALLOW_DECISION = False
#return the output
self.write_output(lp = parameters[0], 
force = self.data[-1, self.FORCE2], 
z = rel_extension)

#run the actual program
r.Receiver(WLCFit).run()

Listing 6: wormlikechain_numerical.py

```python
# coding: utf-8

Created on Wed Mar 29 13:26:44 2017

@author: s115898
@version: 1.0.1

this module can be used to calculate the relative extension of a DNA molecule when a force is applied to it. It is based on the wormlike-chain model. The numerical approximation used here is described in "Stretching DNA" by J. Marko and E. Siggia (1995) all equation references used here are based on that paper Forces used in this module should be in pN and the lengths should be in nm

import numpy as np
from numpy import linalg as LA
from scipy import constants as cns
import scipy.optimize as optimization

def wormlikechain_numerical_version():
    *** returns the version number of the module as a tuple
    ***
    return (1, 0, 2)

def calculate_matrix_element(l, l_acc, f=1, A=1):
    ***Calculate the matrix element l,l'.
```
The matrix element is given by equation 33

Input:

1: value of \( l \)

\( l' \): value of \( l' \)

f: force used for stretching

A: persistence length of the DNA

Output:

**numerical value of the matrix element**

```python
if l == l_acc:
    return l*(l+1)/(2*A)
factor = -f/np.sqrt((2*l+1)*(2*l_acc+1))
if l_acc == (l-1):
    return factor*l
if l == (l_acc-1):
    return factor*l_acc
```

```python
def calculate_matrix(f=1, A=1, size=20):
    """calculate the matrix given by equation 33
    Input:
    f: force used for stretching
    A: persistence length
    size: the size of the matrix, the matrix will be size x size
    Output:
    the matrix representing equation 33
    ""
    matrix = np.zeros((size, size))
    for i in range(0, size, 1):
        matrix[i][i] = calculate_matrix_element(i, i, f, A)
    for i in range(0, size-1, 1):
        matrix[i+1][i] = calculate_matrix_element(i+1, i, f, A)
    for i in range(1, size, 1):
        matrix[i-1][i] = calculate_matrix_element(i-1, i, f, A)
    return matrix
```

```python
def calculate_g(f=1, A=1, cutoff=20):
    """returns the value of g using eq. 33
    Input:
    f: force used for stretching
    A: persistence length
    cutoff: size of the matrix used to calculate g. Use higher
    \[\rightarrow\] cutoff for
    more precision
    Output:
    g for a certain value of f and A
    ""
    m = calculate_matrix(f, A, cutoff)
    return LA.eigvalsh(m).min()
```

```python
def derive_g(f=1, A=1, cutoff=20, df=0.001):
    """calculates dg/df at a point f
    returns the derivate of g in respect to f at a certain point
    input arguments:
    f: the point at which to calculate the derivate
    A: the stiffness
    cutoff: the size of the matrix used to calculate g
    df: the size of the region used to calculate the derivative
    ""
    ```
return (-calculate_g(f+2*df, A, cutoff)+8*calculate_g(f+df, A, cutoff)−8*calculate_g(f−df, A, cutoff)+calculate_g(f−2*df, A, cutoff))/(12*df)

def wlc_numerical_extension(forces, persistence_length=40.6, temperature=300, cutoff=100, accuracy=0.01):
    """ calculates the relative extension of the DNA using eq. 13
    Input:
    forces: a numpy array containing the forces in pn for which to calculate
    persistence_length(optional): The persistence length of the dna in nm
    temperature(optional): the temperature at which the stretching occurs
    cutoff(optional): the matrix size used to calculate the extension (see eq 33)
    accuracy(optional): the stepsize used for the derivative of g.
    Output:
    a numpy array containing the relative extensions for the input forces
    ""
    extensions = np.zeros(forces.size)
    kT = (temperature*cns.Boltzmann)*1e21
    for f in range(0, forces.size, 1):
        extensions[f] = -derive_g(forces[f]/kT, persistence_length, cutoff, accuracy)
    return extensions

def wlc_numerical_multi_extension(forces, persistence_length=40.6, tether_count=1, cutoff=100, accuracy=0.01):
    """calculates the numerical value for multi tethered dna
    ""
    return wlc_numerical_extension(forces/tether_count, persistence_length, temperature, cutoff, accuracy)

def wlc_with_noise(forces, persistence_length=40.6, length=1300, temperature=300, tether_count=1, force_noise_amount=0.1, extension_noise_amount=0, cutoff=100, accuracy=0.01):
    """calculated the numerical values for the wlc model and adds noise to the output
    Input:
    forces: a numpy array containing the forces in pn for which to calculate
    length: Length of the dna used for stretching in nm.
    persistence_length: The persistence length of the dna in nm
    temperature: the temperature at which the stretching occurs
force_noise_amount: amount of noise on the force to add in pN,
              → uniform distribution
extension_noise_amount: amount of noise on the extension data
              → to add in nm,
              → uniform distribution
cutoff: the matrix size used to calculate the extension (see eq
              → 33)
accuracy: the stepsize used for the derivative of g.

Output:
   returns a tuple consisting of two numpy array
   → (extension data, force data)

extension = length*wlc
              numerical_multi_extension(forces,
              → persistance_length,
              → temperature,
              → tether_count
              → ,
              → cutoff, accuracy)
f_noise = force_noise_amount*(2*np.random.uniform(size=forces.size)
              → −1)
if extension_noise_amount > 0:
x_noise = extension_noise_amount*(2*np.random.uniform(size=
              → forces.size)−1)
return (extension+x_noise, forces+f_noise)
return (extension, forces+f_noise)
def wlc_interpolation(extension, length=1300, persistence_length=40.6,
              → temperature=300):
    ***Calculates the required force used to reach a certain extension
    → using an
    interpolation formula
    Input:
    extension: numpy array of extensions
    length: length of the dna
    persistence_length: persistence length of the dna
    temperature: temperature
    ***
    relative_extension = extension/length
    kT = (temperature*cns.Boltzmann)*1e21
    factor = persistence_length/kT+1/1500
    #return_value= kT/(4*persistence_length)∗(1/(1−relative_extension
    → )∗2)
    #1+4∗relative_extension−3∗relative_extension∗2
    return_value=(relative_extension+1/(4∗(1-relative_extension)+2)
    → −1/4−3/4∗relative_extension∗2)*1/factor
    # return_value[extension>(length*0.95)]=65
    return return_value

def wlc_interpolation_dimensionless(beta):
    return (beta+1/(4*(1-beta)+2)-1/4-3/4*beta+2)
def wlc_extensible_interpolation(force, length = 1300, persistence_length
              → = 40.6, s = 1500, temperature = 300):
    kT = (temperature*cns.Boltzmann)*1e21
    return (1-np.sqrt(kT/(force*persistence_length)+force/s)*length

def wlc_kt():

def wlc_sum_of_squared_residuals(length, persistence_length, force_measured, extension_measured):
    M = 1.2*wlc_interpolation(extension_measured, length, persistence_length) / persistence_length
    return np.sum((M*wlc_interpolation(extension_measured, length, persistence_length) - force_measured)**2)/(1+M**2)

B.2 Experiment Simulator

Listing 7: main.py

```python
# coding: utf-8

# The main window for the experiment simulator software
# Created on Thu Apr 6 15:22:30 2017

@author: s115898

import sys, time
from PyQt5.QtWidgets import QApplication, QWidget, QVBoxLayout, QHBoxLayout, QShortcut
from PyQt5.QtWidgets import QProgressBar, QLabel, QGroupBox, QAction, QInputDialog
from PyQt5.QtWidgets import QPushButton, QFileDialog, QComboBox, QProgressDialog, QTextEdit, QDialog
from PyQt5.QtCore import QTimer
from PyQt5.QtNetwork import QLocalServer
from PyQt5.QtGui import QKeySequence
from multiprocessing.pool import ThreadPool
sys.path.insert(0, '../Utils/)
import qtmatplotlib as qplot
import data
importer as database
import qtpipes as qpipes
import qtparameter as param
import extras as ex
from extras import WLCFitData
from utils import *
import configparser as config
import glob as g
import shutil as dir
import os
import numpy as np

def nongui(fun):
    # "Decorator running the function in non-gui thread while processing the gui events."
    from multiprocessing.pool import ThreadPool
```python
def wrap(*args, **kwargs):
    pool = ThreadPool(processes=1)
    async = pool.apply_async(fun, args, kwargs)
    while not async.ready():
        async.wait(0.01)
        QApplication.processEvents()
    return async.get()

return wrap

def openFileNameDialog(parent):
    #***shows an open file dialog***
    options = QFileDialog.Options()
    #options |= QFileDialog.DontUseNativeDialog
    fileName, _ = QFileDialog.getOpenFileName(parent,
                                               "load lumicks data",
                                               ";;Other data (*.asc)",
                                               options=options)
    return fileName

class MainWindow(QtWidgets.QMainWindow):
    def __init__(self):
        QtWidgets.QMainWindow.__init__(self)
        #initializes the gui
        self.initMenu()
        self.initUI()
        #create the server for the IPC
        self.initPipeServer()
        self.FD = tdms.FDData()
        self.experiment_is_running = False
        self.experiment_return_count = 0
        self.experiment_start_index = 0
        self.experiment_end_index = 0
        self.output_parameter_list = []
        self.input_parameter_list = []
        self.data_format = [1, 2, 3]
        #for live updating the plot window
        self.show_live_plot = False
        self.pool = ThreadPool(processes=1)
        self.async = self.pool.apply_async(self.update_live_plot)
        #some extras
        self.wlc_plotter = ex.WLCPlotData()
        #some variables for running a directory
        self.is_running_directory = False
        self.is_running_database = False
        self.file_queue = []
        #
        self.connected_program_name = None
        self.connected_program_version = None
        #database version
        self.database_version = "0.1"
        self.database_threshold = None
        self.data_running_training = False
        self.database_file = None
        #open matplottest window
```

The code snippet contains a function `wrap` that wraps a function with a pool of threads to perform asynchronous operations. It also includes a function `openFileNameDialog` for opening a file dialog, and a class `MainWindow` that initializes the GUI and sets up various variables and functions for running a directory, database version, and other related tasks.
def initMenu(self):
    save_file = QAction("&save FD-curve", self)
    save_file.triggered.connect(self.extras_save_curve)
    execute_directory = QAction("&run directory", self)
    execute_directory.triggered.connect(self.run_directory_open)
    execute_database = QAction("&run database", self)
    execute_database.triggered.connect(self.run_database_open)
    add_wlc_plot = QAction("&add wlc plot", self)
    add_wlc_plot.triggered.connect(self.extras_add_wlc_plot)
    clear_wlc_plot = QAction("&clear wlc plots", self)
    clear_wlc_plot.triggered.connect(self.extras_clear_wlc_plot)
    fit_iwlc = QAction("&fit inextensible wlc", self)
    fit_iwlc.triggered.connect(self.extras_fit_iwlc)
    fit_ewlc = QAction("&fit extensible wlc", self)
    fit_ewlc.triggered.connect(self.extras_fit_ewlc)
    create_result_video = QAction("&Create result video", self)
    create_result_video.triggered.connect(self.extras_create_video)
    print_info = QAction("&Show info", self)
    print_info.triggered.connect(self.extras_show_info)
    font_size = QAction("&Set plot font size", self)
    font_size.triggered.connect(self.extras_font_size)
    performance_graph = QAction("&database performance graph", self)
    performance_graph.triggered.connect(self.extras_show_performance_graph)

    mainMenu = self.menuBar()
    fileMenu = mainMenu.addMenu('&File')
    fileMenu.addAction(save_file)
    fileMenu.addAction(execute_directory)
    fileMenu.addAction(execute_database)
    extraMenu = mainMenu.addMenu('&Extras')
    extraMenu.addAction(add_wlc_plot)
    extraMenu.addAction(fit_iwlc)
    extraMenu.addAction(fit_ewlc)
    extraMenu.addAction(clear_wlc_plot)
    extraMenu.addAction(create_result_video)
    extraMenu.addAction(print_info)
    extraMenu.addAction(font_size)
    extraMenu.addAction(performance_graph)

    #shortcut for starting experiments
    #temp = QAction(self)
    #temp.triggered.connect(self.startExperiment)
    #temp.setShortcut('Ctrl+Q')

    shortcut = QShortcut(QKeySequence("F5"), self)
    shortcut.activated.connect(self.startExperiment)

    shortcut = QShortcut(QKeySequence("Ctrl+O"), self)
    shortcut.activated.connect(self.loadData)

    def initPipeServer(self):
        ""
        creates two pipes for communicating with the clien program
        ""
        print("starting server")
        self.server = QLocalServer()
        self.server.return = QLocalServer()
# creates two invalid pipes that will be updated when a connection is made
self.pipe = qpipes.qt.pipe(0, False)
self.pipe.return = qpipes.qt.pipe(0, False)
# listen for a connection by the program on both pipes
if self.server.listen("lumicks"):
    print("server is listening on: " + self.server.fullServerName())
    self.server.newConnection.connect(self.on_connect)
if self.server.return.listen("lumicks_return");
    print("return server is listening on: " + self.server.return.
    fullServerName();)
    self.server.return.newConnection.connect(self.
    on_connect_return)

def on_connect(self):
    """called when a connection is made on the write pipe""
    print("connection made")
    # accept the connection
    self.socket = self.server.nextPendingConnection()
    # create the pipe
    self.pipe.connect_socket(self.socket)
    self.pipe.read_function(self.handle_input_empty)
    # send the first two commands as a handshake
    self.pipe.write("identify
")
    self.pipe.write("get −output
")

def on_connect_return(self):
    """called when a connection is made on the return pipe""
    print("return connection made")
    self.socket = self.server.return.nextPendingConnection()
    self.pipe.return.connect_socket(self.socket)
    self.pipe.return.read_function(self.handle_input)

def handle_input_empty(self, string):
    """called when input has been received on the write pipe. this shouldn't happen. added for debug purposes""
    print(string)

def handle_input(self, string):
    """handles return commands""
    self.arguments = string.split(" ")
    if(self.arguments[0] == "identify");
        self.program_label.setText((self.arguments[self.arguments.
        index("−name") + 1] + " +
        self.arguments[self.arguments.
        index("−version") + 1]);)
        self.connected_program_name = self.arguments[arguments.
        .index("−name") + 1]
        self.connected_program_version = self.arguments[arguments.
        index("−version") + 1]
        # print(self.arguments)
    if(self.arguments[0] == "output");
        print("adding widget")
        parameter = param.QtParameter(self.outputpanel)
        self.output_layout.addWidget(parameter)
parameter.set_name(self.arguments[2])

self.output_parameter_list.append(parameter)

if(self.arguments[0] == "input"):
    parameter = param.QtInputParameter(self.inputpanel,self.
    self.output_parameter_list.append(parameter)
    self.input_layout.addWidget(parameter)
    self.output_parameter_list.append(parameter)

##automatically set it to the correct configuration
self.set_configuration_from_current_data()

if(self.arguments[0] == "return_data"):
    #increases the data return count by 1
    self.experiment_return_count += 1
    index = 1
    time_stamp = float(self.arguments[-1])
    #set the output in the corresponding widget
    for p in self.output_parameter_list:
        if index < len(self.arguments):
            p.set_value(float(self.arguments[index]),time_stamp
            index = index + 1
    #calculates the performance values
    self.data_counter.set_value(self.experiment_return_count,
    self.experiment_tick < self.FD.time.size:
    self.delay_measure.set_value(self.FD.time[self.
    experiment_tick],time_stamp)

"""if(self.arguments[0] == "format"):
    for n in range(1,len(self.arguments)):
        self.data_format += [int(self.arguments[n])]

    print(self.data_format)"""
def get_output_list(self):
    """returns a list of the output parameters of the input program
    """
    return_list = []
    for p in self.output_parameter_list:
        return_list += [p.name()]
    return return_list

def initUI(self):
    """lays out the main ui for the program"""
    self.setGeometry(300, 300, 300, 220)
    self.setWindowTitle('Experimental simulator')

    #the main widget contains all the contents
    self.main_widget = QWidget(self)

    #widget containing the plot and the parameter area
    self.main_widget = QWidget(self.main_widget.top)
    self.experiment_progress_widget = QProgressBar(self.
    main_widget_top)
self.experiment_progress_widget.setMinimum(0)
self.experiment_progress_widget.setMaximum(10)
self.experiment_progress_widget.setValue(0)

# creates a layout for the info and the plot panel
lmain = QHBoxLayout(self.main_widget)
infopanel = QWidget(self.main_widget)
plotpanel = QWidget(self.main_widget)
lmain.addWidget(infopanel)
lmain.addWidget(plotpanel)

# fill the info panel
l2 = QVBoxLayout(infopanel)
loadButton = QPushButton('Open Data', infopanel)
loadButton.clicked.connect(self.loadData)
self.playButton = QPushButton('Start', infopanel)
self.playButton.clicked.connect(self.startExperiment)
self.startTime = param.QtInputParameter(group_time, None)
self.startTime.set_name("start time")
self.endTime = param.QtInputParameter(group_time, None)
self.endTime.set_name("end time")
self.startTime.varvalue.valueChanged.connect(self.
    time_range_changed)
self.endTime.varvalue.valueChanged.connect(self.
    time_range_changed)

group_time_layout.addWidget(self.startTime)
group_time_layout.addWidget(self.endTime)

# button to export the results
self.expButton = QPushButton('Save Results', infopanel)
self.expButton.clicked.connect(self.save_results)

self.discButton = QPushButton('Disconnect', infopanel)
self.discButton.clicked.connect(self.disconnect)

self.program_label = QLabel('No connection', infopanel)

# groupbox for input parameter
self.inputpanel = QGroupBox("input parameters", infopanel)
self.input_layout=QVBoxLayout(self.inputpanel)

# groupbox for output parameter
self.outputpanel = QGroupBox("output parameters", infopanel)
self.output_layout=QVBoxLayout(self.outputpanel)

# groupbox for the performance measures
self.performancebox=QGroupBox("performance", infopanel)
self.performance_layout=QVBoxLayout(self.performancebox)
self.data_counter=param.QtParameter(self.performancebox)
self.data_counter.set_name("counter")
self.delay_measure=param.QtParameter(self.performancebox)
self.delay_measure.set_name("delay(ms)")

self.performance_layout.addWidget(self.data_counter)
self.performance_layout.addWidget(self.delay_measure)

# add everything to the infopanel layout
l2.addWidget(loadButton)
l2.addWidget(self.playButton)
l2.addWidget(self.program_label)
l2.addWidget(group_time)
l2.addWidget(self.inputpanel)
l2.addWidget(self.outputpanel)
l2.addWidget(self.performancebox)
l2.addWidget(self.expButton)
l2.addWidget(self.discButton)
l2.setAlignment(QtCore.Qt.AlignTop)

# filling the plot widget
l = QVBoxLayout(plotpanel)
combobox to change the plot type
self.plottype = QComboBox(plotpanel)
self.plottype.addItems(
    ["force,extension","time,distance","bead movement",
     "angle, force1", "angle, force2"]
)
sel.currentplot = 0
self.plottype.currentIndexChanged.connect(self.change_plot)

## the actual plot window
self.plot = qplot.mpl_widget(plotpanel)
l.addWidget(self.plottype)
l.addWidget(self.plot)
# l.addWidget(NavigationToolbar(self.plot, plotpanel))

## create the layout for the main panels and the progress bar
lprogress = QVBoxLayout(self.main_widget_top)
lprogress.addWidget(self.main_widget)
lprogress.addWidget(self.experiment_progress_widget)
self.setCentralWidget(self.main_widget_top)
self.showMaximized()
self.show()

def disconnect(self):
    
    "disconnect the two pipes if they are connected""
    self.pipe.close()
    self.pipe_return.close()
    removing the output widgets
    for p in self.output_parameter_list:
        self.output_layout.removeWidget(p)
        p.setParent(None)
    self.outputpanel.repaint()
    self.output_parameter_list.clear()
    # removing the input parameter widgets
    for p in self.input_parameter_list:
        self.input_layout.removeWidget(p)
        p.setParent(None)
    self.inputpanel.repaint()
    self.input_parameter_list.clear()
    # clearing the data for the performance widgets
    self.data_counter.clear()
    self.delay_measure.clear()
    self.program_label.setText("No connection")
    self.connected_program_name = None
def loadData(self):
    """ opens the tdms/asc file using a file dialog""
    self.file_to_load = openFileNameDialog(self)
    if len(self.file_to_load) == 0:
        return
    self.load_data_from_file(self.file_to_load)

def load_data_from_file(self,fname):
    """load the data form the tdms file and shows a progress dialog
    ""
    #store the filename in a variable
    self.file_to_load = fname
    #show a progress bar
    self.progress = QProgressDialog("Loading data","cancel", 0, 0,
    self)
    self.progress.setWindowModality(QtCore.Qt.WindowModal)
    self.progress.setCancelButton(None)
    self.progress.forceShow()
    #here the actual file is being loaded in a different function
    #to make sure the gui is not being blocked
    self.loadDataActual()
    self.progress.close()
    #update the time range
    self.endTime.set_range(self.FD.time[0]/1000,
    self.FD.time[1]/1000)
    self.startTime.set_range(self.FD.time[0]/1000,
    self.FD.time[1]/1000)
    self.endTime.set_value(self.FD.time[0]/1000)
    self.startTime.set_value(self.FD.time[0]/1000)
    #default to the full time range
    self.experiment_start_index=0
    self.experiment_end_index=self.FD.time.size
    #set the configuration
    self.set_configuration_from_current_data()

@nongui
def loadDataActual(self):
    """ loads the actual data from the file and updates the plot
    this function can be called without the gui being blocked""
    self.FD.load(self.file_to_load)
    self.change_plot(self.currentplot)

def startExperiment(self):
    """called when the experiment is started""
    if self.experiment_is_running == True:
        return self.stopExperiment()
    if self.FD.valid == False:
        return
    for inputs in self.input_parameter_list:
        inputs.value_change()
    self.pipe.write("start {}
    self.experiment_timer = QTimer()
self.experiment_timer.timeout.connect(self.send_data)
self.experiment_timer.start(self.FD.time[1] - self.FD.time[0])
self.experiment_tick = self.experiment_start_index
self.experiment_is_running = True
self.experiment_progress_widget.setMaximum(self.
  experiment_end_index -
  self.
  experiment_start_index
  )

self.playButton.setText("Stop")
#reset all parameter data
for p in self.output_parameter_list:
p.clear()
#reset the performance measurements
self.experiment_return_count = 0
self.data_counter.clear()
self.delay_measure.clear()

def stopExperiment(self):
    """stops the experiment""
    self.update_live_plot()
    self.experiment_timer.stop()
    self.experiment_is_running = False
    self.experiment_progress_widget.setValue(0)
    self.playButton.setText("Start")
    self.pipe.write("stop\n")
    #this will start the next file in the queue
    if self.is_running_directory:
        self.run_next_file()
        
    def change_plot(self,i, show_legend = True):
        """changes the plot in the main plot window in the center of
        the screen
        the input i is the integer which signals the index of the
        combobox on
        top of the plot area
        """
        self.currentplot = self.plottype.currentIndex()
        if self.FD.valid == False:
            #no file was loaded
            return
        if i == 0:
            #plot force distance
            self.plot.axes().plot(self.FD.d[self.experiment_start_index:
            self.FD.d[self.experiment_end_index-1]) - self.FD.bead_size,
            self.FD.f1[self.experiment_start_index:self.
            self.experiment_end_index-1],label="trap 1")
            self.plot.axes().plot(self.FD.d[self.experiment_start_index:
            self.FD.d[self.experiment_end_index-1]) - self.FD.bead_size,
            self.FD.f2[self.experiment_start_index:self.
            self.experiment_end_index-1],label="trap 2")
            self.plot.XY_label(r"distance($\mu$m)$","force(pN)"")
            
            #adding the plots
for i in range(0, self.wlc_plotter.count()):
    x, f = self.wlc_plotter.get_plot(i)
    self.plot.axes().plot(x, f, label=self.wlc_plotter.label(i))

if show_legend:
    self.plot.axes().legend()
self.plot.redraw()

# creating the live plots
if self.show_live_plot:
    self.live_plot1 = self.plot.create_live_plot()
    self.live_plot2 = self.plot.create_live_plot()

elif i == 1:
    # plot distance time
    self.plot.full_clear()
    self.plot.axes().plot(self.FD.time[:self.
        experiment_start_index: (self.experiment_end_index - 1)
        ], self.FD.d[:self.experiment_start_index: (self.
        experiment_end_index - 1)] - self.FD.bead_size)
    self.plot.XY_label("time(ms)", "distance(\(\mu\)m)"")
    self.plot.redraw()

elif i == 2:
    # plot bead movement
    self.plot.full_clear()
    self.plot.axes().plot(self.FD.b1x[:self.
        experiment_start_index: (self.experiment_end_index - 1)
        ], self.FD.b1y[:self.experiment_start_index: (self.
        experiment_end_index - 1)], label='bead 1')
    self.plot.axes().plot(self.FD.b2x[:self.
        experiment_start_index: (self.experiment_end_index - 1)
        ], self.FD.b2y[:self.experiment_start_index: (self.
        experiment_end_index - 1)], label='bead 2')
    self.plot.XY_label("x position(\(\mu\)m)", "y position(\(\mu\)m)"")
    self.plot.axes().legend()
    self.plot.redraw()

elif i == 3:
    # custom plot
    self.plot.full_clear()
    ##self.plot.axes().plot(self.FD.d[:self.
        experiment_start_index: (self.experiment_end_index - 1)
        ] - self.FD.bead_size,
    ##inp[:self.experiment_start_index: (self.experiment_end_index - 1)])
    ##self.plot.axes().hist(np.rad2deg(np.arccos(inp1[:self.
        experiment_start_index: (self.experiment_end_index - 1)])), 40, label='trap 1')
    ##self.plot.axes().hist(np.rad2deg(np.arccos(inp[:self.
        experiment_start_index: (self.experiment_end_index - 1)])), 40, label='trap 2')

    angles = np.rad2deg(np.arctan2(self.FD.f1y, self.FD.f1x))
    import matplotlib.pyplot as plt
    cm = plt.cm.get_cmap('jet')
    sc = self.plot.axes().scatter(angles[:self.
        experiment_start_index: (self.experiment_end_index - 1)],
        self.FD.f1[:self.
            experiment_start_index: (self.experiment_end_index - 1)],
        c='r', norm=cm)
    self.plot.redraw()
def update_live_plot(self):
    
    """this updates the live plot: a small red overlay indicating in realtime
    """
    if self.show_live_plot == False:
        return
    if self.FD.valid == False:
        #no file was loaded
        return
    start_index = self.experiment_tick-50
    if start_index < self.experiment_start_index:
        start_index = self.experiment_start_index
    end_index=self.experiment_tick
    if self.currentplot == 0:
        angles = np.rad2deg(np.arctan2(self.FD.f2y,self.FD.f2x))
        f2 = np.sqrt(self.FD.f2x**2+self.FD.f2y**2)
        import matplotlib.pyplot as plt
        cm = plt.cm.get_cmap('jet')
        sc = self.plot.axes().scatter(angles[self.
        \experiment_start_index:(self.experiment_end_index-1)],
        f2[self.
        \experiment_start_index ::(self.
        \experiment_end_index-1)],
        c = self.FD.time[self.
        \experiment_start_index ::(self.
        \experiment_end_index-1)],
        cmap = cm)
        cbar = self.plot.plot.fig.colorbar(sc)
        cbar.set_label('time(ms)')
        self.plot.XY_label("angle(degrees)","force(pN)")
        self.plot.redraw()

def update_live_plot(self):
    
    """this updates the live plot: a small red overlay indicating in realtime
    """
    if self.show_live_plot == False:
        return
    if self.FD.valid == False:
        #no file was loaded
        return
    start_index = self.experiment_tick-50
    if start_index < self.experiment_start_index:
        start_index = self.experiment_start_index
    end_index=self.experiment_tick
    if self.currentplot == 0:
        angles = np.rad2deg(np.arctan2(self.FD.f2y,self.FD.f2x))
        f2 = np.sqrt(self.FD.f2x**2+self.FD.f2y**2)
        import matplotlib.pyplot as plt
        cm = plt.cm.get_cmap('jet')
        sc = self.plot.axes().scatter(angles[self.
        \experiment_start_index:(self.experiment_end_index-1)],
        f2[self.
        \experiment_start_index ::(self.
        \experiment_end_index-1)],
        c = self.FD.time[self.
        \experiment_start_index ::(self.
        \experiment_end_index-1)],
        cmap = cm)
        cbar = self.plot.plot.fig.colorbar(sc)
        cbar.set_label('time(ms)')
        self.plot.XY_label("angle(degrees)","force(pN)")
        self.plot.redraw()
send_data(self):
    
    """sends the data to the analysis program""
    
    # write the information at the current position to the pipe
    data_string = "data {}:{}\n".format(self.FD.time[self.experiment_tick])
    for i in range(1,12):
        data_string += " {}:{}\n".format(self.FD.get_value(i,self.experiment_tick))
    
    self.pipe.write(data_string+"\n")
    
    """self.pipe.write("data {}:{} {}:{} {}:
    self.FD.f2[{}],
    self.FD.d[{}])""

    # go to the next position in the pipe
    self.experiment_tick += 1
    if self.async.ready():
        self.async = self.pool.apply_async(self.update_live_plot)

    self.experiment_progress_widget.setValue(self.experiment_tick-
    self.experiment_start_index)
    self.experiment_progress_widget.setFormat("time: {}:
    self.FD.time[{}-1]/1000)

    if last position has been send: stop the experiment
    if(self.experiment_tick == self.experiment_end_index):
        self.experiment_tick -= 1
        self.stopExperiment()

save_results(self):
    """stores the results in an asc file""
    filename, _ = QFileDialog.getSaveFileName(self,"save results", "
    ","asc-file (*.asc)")
    if len(filename) == 0:
        return
    self.save_results_to_file(filename)

save_results_to_file(self, fname):
    """writes the results to a file""
    
    # define a columns for the file and the data which belong in
    each column
    header = ["time"]# always write time
    data = [self.data_counter.time_array()]
    # now add the input parameters to the file
    for p in self.output_parameter_list:
        header += [p.name()]
        data += [p.value_array()]
#write performance measures
info += ['analysis program version: ' + self.connected_program_version]
if self.database_version is not None:
    info += ['database version: ' + self.database_version]
header += ['count', 'delay']
data += [self.data_counter.value_array(), self.delay_measure.value_array()]
#write the data to the file
save_data_asc(fname, data, header, info)

#time range changed
"""this function gets called when one of the two double spinboxes governing the start and end time of the experiments get changed"""
if self.FD.valid == False:
    #no file was loaded
    return
#convert the time in in spinboxes to indexes in the arrays
if(self.endTime.get_value() <= self.startTime.get_value()):
    self.endTime.set_value(self.startTime.get_value())
self.experiment_start_index = 0
self.experiment_end_index = 0
start_list = np.where((self.FD.time < (self.startTime.get_value() * 1000)) == True)[0]
if start_list.size > 0:
    self.experiment_start_index = start_list[-1]
end_list = np.where((self.FD.time < (self.endTime.get_value() * 1000)) == True)[0]
if end_list.size > 0:
    self.experiment_end_index = end_list[-1]+1
#update the plot to reflect the changes
self.change_plot(self.currentplot)

#extras add wlc plot
lc, okPressed = QInputDialog.getDouble(self, "wlc plot", "contour length:", 11, 0, 999999, 10)
if not okPressed:
    return
lp, okPressed = QInputDialog.getDouble(self, "wlc plot", "persistence length:", 50, 0, 999999, 10)
if not okPressed:
    return
S, okPressed = QInputDialog.getDouble(self, "wlc plot", "stretch modulus:", 0, 0, 999999, 10)
if not okPressed:
    return
self.wlc_plotter.add_plot(lc, lp, S)
sself.change_plot(self.currentplot)

#extras clear wlc plot
self.wlc_plotter.clear()
sself.change_plot(self.currentplot)

#extras fit iwlc
self.wlc_plotter.clear()
sself.change_plot(self.currentplot)
629 \[ d = self.FD.d[self.experiment\_start\_index:(self.\→ experiment\_end\_index−1)]−self.FD.bead\_size \]
630 \[ f = self.FD.f2[self.experiment\_start\_index:(self.\→ experiment\_end\_index−1)] \]
631 \[ result = WLCFitData.wlc\_interpolation\_fit(d*1000 , f) \]
632 if result is not None:
633 \[ self.wlc\_plotter.add\_plot(result[0]/1000, result[1]) \]
634 \[ self.change\_plot(self.currentplot) \]
635
636 def extras\_fit\_wlc(self):
637 \[ d = self.FD.d[self.experiment\_start\_index:(self.\→ experiment\_end\_index−1)]−self.FD.bead\_size \]
638 \[ f = self.FD.f2[self.experiment\_start\_index:(self.\→ experiment\_end\_index−1)] \]
639 \[ result = WLCFitData.wlc\_extensible\_fit(d , f) \]
640 if result is not None:
641 \[ self.wlc\_plotter.add\_plot(result[0], result[1], result[2]) \]
642 \[ self.change\_plot(self.currentplot) \]
643
644 def extras\_create\_video(self):
645 \[ print("creating video") \]
646 temp\_index = self.experiment\_end\_index
647 FPS = 30
648 \[ time\_between\_data\_points = (self.FD.time[1]−self.FD.time[0])\→ /1000 \]
649 \[ data\_points\_per\_frame = int((1/FPS)/(time\_between\_data\_points)) \]
650 for i in range(self.experiment\_start\_index+1,self.\→ experiment\_end\_index,data\_points\_per\_frame):
651 \[ self.experiment\_end\_index = i \]
652 \[ xt = self.plot.\_axes().\_get\_xlim() \]
653 \[ yt = self.plot.\_axes().\_get\_ylim() \]
654 \[ self.change\_plot(self.currentplot, False) \]
655 \[ self.plot.\_axes().\_set\_xlim(*xt) \]
656 \[ self.plot.\_axes().\_set\_ylim(*yt) \]
657 \[ self.plot.\_save("Video\plot{d}.png".\_format(i)) \]
658 \[ self.experiment\_end\_index = temp\_index \]
659 for p in self.output\_parameter\_list:
660 \[ test\_range = range(self.experiment\_start\_index+1,self.\→ experiment\_end\_index,data\_points\_per\_frame) \]
661 \[ p.create\_video(self.FD.time[test\_range]) \]
662
663 def extras\_show\_info(self):
664 if not self.FD.is\_valid():
665 \[ return \]
666 dialog=QDialog(self)
667 lmain= QBoxLayout(dialog)
668 info\_text=QTextEdit(dialog)
669 info\_text.set\_text(self.FD.info())
670 info\_text.set\_read\_only(True)
671 lmain.\_add\_widget(info\_text)
672 dialog.\_set\_window\_title("File info")
673 dialog.\_set\_window\_modality(QtCore.Qt.\_non\_modal)
674 dialog.\_set\_window\_flags(dialog.\_window\_flags() \| QtCore.Qt.\_window\_system\_menu\_hint \| QtCore.Qt.\_window\_min\_max\_buttons\_hint)
675 dialog.\_show()
dialog.activateWindow()

def extras_font_size(self):
    """Creates an input dialog which asks the user for a plot font size. It will set this font size for all plots""
    lc, okPressed = QInputDialog.getDouble(self, "plot font", "fontsize",
                                           16, 0, 999999, 10)
    if not okPressed:
        return
    qplot.setFont(int(lc))
    self.plot.full_clear()
    self.change_plot(self.currentplot)

def extras_save_curve(self):
    """stores the FD-curve in an tdms file""
    filename, = QFileDialog.getSaveFileName(self, "save FD-curve",
                                              "tdms-file (*.tdms)")
    if len(filename) == 0:
        return
    self.FD.save(filename, self.experiment_start_index, (self.experiment_end_index - 1))

def run_directory_open(self):
    """this function allows the user to select a directory and run every file in this directory. The results will be save automatically""
    if self.connected_program_name is None:
        print("please connect a program first")
        return
dir_ = QFileDialog.getExistingDirectory(self, 'Select a folder: ', '', QFileDialog.ShowDirsOnly)
    if len(dir_) <= 0: return
    print("running directory: " + dir_)
    for file in g.glob(dir_ + "\*.tdms", recursive=True):
        dirname = os.path.dirname(os.path.abspath(file)) + "\results" + self.connected_program_name
        if not os.path.exists(dirname):
            os.makedirs(dirname)
        self.file_queue += [file]
    self.is_running_directory = True
    self.run_next_file(save_current_results = False)

def run_database_open(self):
    """this function allows the user to open a database and the entire database will be run and the results will be save automaticcally""
    if self.connected_program_name is None:
        print("please connect a program first")
        return
database_file, = QFileDialog.getOpenFileName(self, "open database", "database-file (*.ini)")
    if len(database_file) <= 0: return
    self.database_file = database_file
dir_ = os.path.dirname(database_file)
c = config.ConfigParser()
c.read(database_file)
curve_types = c.options("CurveDirectories")
self.database_version = c.get("Info","Database Version")
np.random.seed(c.getint("Test Set Info","Random Seed"))
self.database_threshold = c.getfloat("Test Set Info","Threshold")

def run_next_file(self, save_current_results = True):
    if save_current_results:
        if self.data_running_training:
            output_dir = os.path.dirname(os.path.abspath(self.FD.
                "file"))+ "\\results"+self.connected_program_name
            output_file = output_dir+"\\"+os.path.basename(self.FD.
                "file")+".asc"
            self.save_results_to_file(output_file)
        else:
            self.save_results_to_file(output_file)
```python
output_dir = os.path.dirname(os.path.abspath(self.FD.file)) + '/results_test' + self.connected_program_name
output_file = output_dir + os.path.basename(self.FD.file) + '.asc'
self.save_results_to_file(output_file)

if len(self.file_queue) > 0:
    #run the next file
    next_file = self.file_queue.pop()
    random_value = np.random.rand()
    if random_value < self.database_threshold and self.data_running_training:
        self.start_running_file(next_file)
    elif random_value >= self.database_threshold and not self.data_running_training:
        self.start_running_file(next_file)
    else:
        self.run_next_file(save_current_results = False)
else:
    """finished running the current set of files""
    if (self.data_running_training is False) and (self.is_running_database):
        #show performance graph since a test set was just used
        self.show_performance_graph(self.database_file)

self.is_running_directory = False
self.is_running_database = False
self.database_version = None
self.database_file = None

def start_running_file(self, filename):
    self.load_data_from_file(filename)
    self.startExperiment()

def set_configuration_from_current_data(self):
    """update the current input boxes, with the configuration stored in the .config file with the tdms file""
    if not self.FD.is_valid: return
    for inputs in self.input_parameter_list:
        if inputs.name() in self.FD.config:
            inputs.set_value(self.FD.config[inputs.name()])

@nongui
def calc_performance_graph(self, file):
    print("calculating performance graph")
    db = database.database(file)
    test_threshold = np.linspace(0, 1, 10)
    results = db.get_performance_result("1 molecule", self.connected_program_name, test_threshold)
    return test_threshold, results

def show_performance_graph(self, file):
    self.progress = QProgressDialog("Calculating performance", ", cancel", 0, 0, self)
    self.progress.setWindowModality(QtCore.Qt.WindowModal)
```

58
self.progress.setCancelButton(None)
self.progress.forceShow()
test_threshold, results = self.calc_performance_graph(file)
self.progress.close()

dialog=QDialog(self)
plot=qplot.mpl_widget(dialog)
lmain= QHBoxLayout(dialog)
from matplotlib.font_manager import FontProperties
fontP = FontProperties()
fontP.set_size('small')
"""create the plot""
ax = plot.axes()
plot.set_font_size(20)
for key,value in results.items():
    if key != 'loss':
        ax.plot(test_threshold, value*100, label = key+ 'false positives')
    else:
        ax.plot(test_threshold, value*100, label = 'false negatives')
plot.XYlabel("threshold probability","fraction(%)")
#setting the font
art = ax.legend(bbox_to_anchor=(1,1), prop = fontP)
lmain.addWidget(plot)
dialog.setWindowTitle("Dialog")
dialog.setWindowModality(QtCore.Qt.NonModal)
dialog.setWindowFlags(dialog.windowFlags() |
                        QtCore.Qt.WindowSystemMenuHint |
                        QtCore.Qt.WindowMinMaxButtonsHint)
dialog.show()

def extras_show_performance_graph(self):
database_file, = QFileDialog.getOpenFileName(self,"open database", 
                                           "database-file (*.ini)")
if len(database_file) <= 0: return
self.show_performance_graph(database_file)

if __name__ == '__main__':
    app = QApplication(sys.argv)
ex = MainWindow()
sys.exit(app.exec_())

Listing 8: data_importer.py

# ←← coding: utf-8 ←←
***
Created on Mon Apr  3 15:36:48 2017

59
import tdmsutils
from nptdms import TdmsWriter, RootObject, GroupObject, ChannelObject
import os
import scipy.io as sio

class FDData:
    """Class to store the force extension data
    """
    def __init__(self, fname=""):
        self.file = ""
        self.config = {}
        if len(fname)>0 :
            self.load(fname)
        else:
            self.valid = False

    def is_valid(self):
        return self.valid

    def info(self):
        if self.tdms is not None:
            return tdmsutils.text_tdms_structure(self.tdms, True, False

    def save(self,fname, start_index = 0, end_index = -1):
        if not self.valid: return
        root_object = RootObject(properties={
            "prop1": "foo",
            "prop2": 3,
        })
        root_object = self.tdms.object()
        group_object = GroupObject("group 1", properties={
            "prop1": 1.2345,
            "prop2": False,
        })
        group_object = self.tdms.object("FD Data")
        #data = np.array([1.0, 2.0, 3.0, 4.0, 5.0])
        with TdmsWriter(fname) as tdms_writer:
            def write_data(name, data):
                channel_object = self.tdms.object("FD Data", name)
                write_object = ChannelObject("FD Data", name, data, 
                channel_object.properties)
                tdms_writer.write_segment([
                    root_object,
                    group_object,
                    write_object])

            write_data("Time (ms)", self.time[start_index:end_index])
```python
write_data("Force Trap 0 (pN)", self.f1[start_index:end_index])
write_data("Force Trap 1 (pN)", self.f2[start_index:end_index])
write_data("Distance 1 (um)", self.d[start_index:end_index])
write_data("Force Channel 0 (pN)", self.f1x[start_index:end_index])
write_data("Force Channel 1 (pN)", self.f1y[start_index:end_index])
write_data("Force Channel 2 (pN)", self.f2x[start_index:end_index])
write_data("Force Channel 3 (pN)", self.f2y[start_index:end_index])
write_data("Bead 1 X position (um)", self.b1x[start_index:end_index])
write_data("Bead 1 Y position (um)", self.b1y[start_index:end_index])
write_data("Bead 2 X position (um)", self.b2x[start_index:end_index])
write_data("Bead 2 Y position (um)", self.b2y[start_index:end_index])
write_data("Force Channel 0 STDEV (pN)", self.f1x_std[start_index:end_index])
write_data("Force Channel 1 STDEV (pN)", self.f1y_std[start_index:end_index])
write_data("Force Channel 2 STDEV (pN)", self.f2x_std[start_index:end_index])
write_data("Force Channel 3 STDEV (pN)", self.f2y_std[start_index:end_index])
write_data("Force Trap 0 STDEV (pN)", self.f1_std[start_index:end_index])
write_data("Force Trap 1 STDEV (pN)", self.f2_std[start_index:end_index])

def load(self, fname):
    self.valid = True
    self.tdms = None
    if fname.endswith('.tdms') == True:
        self.file = fname
        tdmsFile = tdmsutils.TdmsFile(fname)
        #print(tdmsutils.text_tdms_structure(tdmsFile,True,False))
        #tdmsutils.print_tdms_structure(tdmsFile,True,False)
        self.time=tdmsFile.object("FD Data", 'Time (ms)').data
        self.f1 = tdmsFile.object("FD Data", 'Force Trap 0 (pN)').data
        self.f2 = tdmsFile.object("FD Data", 'Force Trap 1 (pN)').data
        self.d = tdmsFile.object("FD Data", 'Distance 1 (um)').data
        self.f1x= tdmsFile.object("FD Data", 'Force Channel 0 (pN)').data
        self.f1y= tdmsFile.object("FD Data", 'Force Channel 1 (pN)').data
        self.f2x= tdmsFile.object("FD Data", 'Force Channel 2 (pN)').data
        self.f2y= tdmsFile.object("FD Data", 'Force Channel 3 (pN)').data
```

self.f2y = tdmsFile.object('FD Data', 'Force Channel 3 (pN)').data
self.b1x = tdmsFile.object('FD Data', 'Bead 1 X position (um)').data
self.b1y = tdmsFile.object('FD Data', 'Bead 1 Y position (um)').data
self.b2x = tdmsFile.object('FD Data', 'Bead 2 X position (um)').data
self.b2y = tdmsFile.object('FD Data', 'Bead 2 Y position (um)').data
self.f1x_std = tdmsFile.object('FD Data', 'Force Channel 0 STDEV (pN)').data
self.f1y_std = tdmsFile.object('FD Data', 'Force Channel 1 STDEV (pN)').data
self.f2x_std = tdmsFile.object('FD Data', 'Force Channel 2 STDEV (pN)').data
self.f2y_std = tdmsFile.object('FD Data', 'Force Channel 3 STDEV (pN)').data

self.f1_std = tdmsFile.object('FD Data', 'Force Trap 0 STDEV (pN)').data
self.f2_std = tdmsFile.object('FD Data', 'Force Trap 1 STDEV (pN)').data

self.bead_size = tdmsFile.object('FD Data').property("Bead Diameter (um)")
self.f1x_stiffness = tdmsFile.object('FD Data', 'Force Channel 0 (pN)').property("Trap stiffness (pN/m)")
self.f1y_stiffness = tdmsFile.object('FD Data', 'Force Channel 1 (pN)').property("Trap stiffness (pN/m)")
self.f2x_stiffness = tdmsFile.object('FD Data', 'Force Channel 2 (pN)').property("Trap stiffness (pN/m)")
self.f2y_stiffness = tdmsFile.object('FD Data', 'Force Channel 3 (pN)').property("Trap stiffness (pN/m)")

## check if a default config file is available
dirname = os.path.dirname(os.path.abspath(fname))
if os.path.exists(dirname + '/default.config'):
    my_file = open(dirname + '/default.config', 'r')
    for line in my_file:
        args = line.split("=")
        if len(args) == 2:
            name = args[0].rstrip().lstrip()
            value = args[1].rstrip().lstrip()
            self.config[name] = float(value)

## check if there is a config file available
if os.path.exists(fname + '.config'):
    my_file = open(fname + '.config', 'r')
    for line in my_file:
        args = line.split("=")
        if len(args) == 2:
            name = args[0].rstrip().lstrip()
            value = args[1].rstrip().lstrip()
            self.config[name] = float(value)

elif fname.endswith('.asc') == True:
self.tempdata, self.tempch = load_lumicks_data_asc(fname)
self.time = self.tempdata[:, self.tempch['time']]
self.d = self.tempdata[:, self.tempch['distance']]
self.f1 = self.tempdata[:, self.tempch['trap1_force']]
self.f2 = self.tempdata[:, self.tempch['trap2_force']]  
self.f1y = self.tempdata[:, self.tempch['trap1_force_y']]
self.f2y = self.tempdata[:, self.tempch['trap2_force_y']]
self.f1x = self.tempdata[:, self.tempch['trap1_force_x']]
self.f2x = self.tempdata[:, self.tempch['trap2_force_x']]
self.b1y = self.tempdata[:, self.tempch['bead1_y']]
self.b2y = self.tempdata[:, self.tempch['bead2_y']]
self.b1x = self.tempdata[:, self.tempch['bead1_x']]
self.b2x = self.tempdata[:, self.tempch['bead2_x']]
self.bead_size = 0.9

e1f fname.endswith('.mat') == True:
    # Load matlab file
    contents = sio.loadmat(fname, squeeze_me=True)
    self.f1 = np.transpose(contents['force1'])
    self.time = np.transpose(contents['time']).astype(float)
    self.d = np.transpose(contents['distance'])
    self.f2 = np.transpose(contents['force2'])
    self.f1x = np.transpose(contents['force1_x'])
    self.f1y = np.transpose(contents['force1_y'])
    self.f2x = np.transpose(contents['force2_x'])
    self.f2y = np.transpose(contents['force2_y'])
    self.b1x = np.transpose(contents['b1x'])
    self.b1y = np.transpose(contents['b1y'])
    self.b2x = np.transpose(contents['b2x'])
    self.b2y = np.transpose(contents['b2y'])
    self.bead_size = float(contents['bead_diameter'])
    self.get_config(fname)
    self.file = fname
    # print(contents)

def get_config(self, fname):
    ## Check if a default config file is available
    dirname = os.path.dirname(os.path.abspath(fname))
    if os.path.exists(dirname + '/default.config'):
        my_file = open(dirname + '/default.config', 'r')
        for line in my_file:
            args = line.split('=', 'r')
            if len(args) == 2:
                name = args[0].rstrip().lstrip()
                value = args[1].rstrip().lstrip()
                self.config[name] = float(value)
    ## Check if there is a config file available
    if os.path.exists(fname + '.config'):
        my_file = open(fname + '.config', 'r')
        for line in my_file:
            args = line.split('=', 'r')
            if len(args) == 2:
                name = args[0].rstrip().lstrip()
                value = args[1].rstrip().lstrip()
self.config[name] = float(value)

def get_value(self, column, index):
    if column == 1:
        return self.d[index]
    elif column == 2:
        return self.f1[index]
    elif column == 3:
        return self.f2[index]
    elif column == 4:
        return self.f1x[index]
    elif column == 5:
        return self.f1y[index]
    elif column == 6:
        return self.f2x[index]
    elif column == 7:
        return self.f2y[index]
    elif column == 8:
        return self.b1x[index]
    elif column == 9:
        return self.b1y[index]
    elif column == 10:
        return self.b2x[index]
    elif column == 11:
        return self.b2y[index]

def load_lumicks_data_asc(filename, split_character='\t'):
    """reads data for a lumicks experiment stored in an asc-file
    the function returns a matrix where each column represent a
    measured value
    Input:
    filename: the name of the file where the data is stored
    split_character: character used to seperate columns in the asc-
    file
    Returns:
    data: the data from the experiment as a matrix
    channeldata: a dictionary describing what value is stored in
    each column
    of the data matrix
    """
    my_file = open(filename)
    # first read the header
    line=my_file.readline()
    header=line.split(split_character)
    time_index=header.index("Time (ms)")
    distance_index=header.index("Distance 1 (um)")
    force1x_index=header.index("Force Channel 1 (pN)")
    force1y_index=header.index("Force Channel 2 (pN)")
    force2x_index=header.index("Force Channel 3 (pN)")
    force2y_index=header.index("Force Channel 4 (pN)")
    force1_trap_index=header.index("Force Trap 1 (pN)")
    force2_trap_index=header.index("Force Trap 2 (pN)")
    bead1x_index=header.index("Bead 1 X position (um)")
    bead1y_index=header.index("Bead 1 Y position (um)")
bead2_x_index=header.index("Bead 2 X position (um)")
bead2_y_index=header.index("Bead 2 Y position (um)")

#start reading the actual data

temp_data=[]
for line in my_file:
    data=line.split(split_character)
    temp_data+=[[float(data[time_index]), float(data[distance_index])],
                float(data[force1_x_index]), float(data[force1_y_index]),
                float(data[force2_x_index]), float(data[force2_y_index]),
                float(data[bead1_x_index]), float(data[bead1_y_index]),
                float(data[bead2_x_index]), float(data[bead2_y_index]),
                float(data[force1_index]), float(data[force2_index])]]

my_file.close()

#create metadata containing information about the channels
channeldata={"time":0, "distance":1,
             "trap1_force_x":2, "trap1_force_y":3,
             "trap2_force_x":4, "trap2_force_y":5,
             "bead1_x":6, "bead1_y":7,
             "bead2_x":8, "bead2_y":9,
             "trap1_force":10, "trap2_force":11}

return np.array(temp_data), channeldata

#fd = FDData("test_no_molecule.mat")
#time = 1000/(fd.time[1]-fd.time[0])

B.3 KDE Creator

Listing 9: interface.py

# -*- coding: utf-8 -*-

Created on Tue May 2 16:11:35 2017

@author: David den Boef

import glob as g
import kde_learner as k
import logistic_regression_learner as log
import gauss_learner as gauss
import numpy as np
import matplotlib.pyplot as plt
import platform
import os.path

from PyQt5 import QtCore, QtWidgets
from PyQt5.QtWidgets import QApplication, QWidget, QVBoxLayout , QHBoxLayout, QTabWidget, QFormLayout
from PyQt5.QtWidgets import QPushButton, QFileDialog, QLabel, QSpinBox, 
QDoubleSpinBox, QLineEdit, QComboBox
from PyQt5.QtWidgets import QCheckBox

import matplotlib.pyplot as plt
import kde_database as database
import data_loading as d

class KDEWidget(QWidget):
    def __init__(self):
        super().__init__()
        self.file_list = []
        self.database = None
        self.initUI()
        self.loglearner = log.LogRegression()
        self.logrange = None

    def initUI(self):
        self.setGeometry(100, 100, 800, 600)
        self.setWindowTitle('KDE')
        # create the two main widget
        panel = QWidget(self)
        tabpanel = QTabWidget(self)
        tab1 = QWidget(tabpanel)
        tab2 = QWidget(tabpanel)
        tabpanel.addTab(tab1, "KDE")
        tabpanel.addTab(tab2, "Logistic Regression")
        self.plot_kde = qplot.mpl_widget(tab1)
        self.plot_mask = qplot.mpl_widget(tab2)
        hlayout = QHBoxLayout(self)
        hlayout.addWidget(panel)
        hlayout.addWidget(tabpanel)
        # creating the tab layout
        vlayout = QVBoxLayout(tab1)
        vlayout.addWidget(self.plot_kde)
        vlayout = QVBoxLayout(tab2)
        vlayout.addWidget(self.plot_mask)
        # creating all the widgets in the panel
        self.open_directory = QPushButton("Open database", panel)
        self.create_kde_button = QPushButton("Create KDE", panel)
        self.update_kde_button = QPushButton("Update KDE", panel)
        self.create_log_button = QPushButton("Create log regression", panel)
        self.combo_curve_type = QComboBox(panel)
        self.combo_curve_type.currentIndexChanged.connect(self.
        change_curve_type)
73 self.combo_analysis_module_type = QComboBox(panel)
74 self.combo_analysis_module_type.currentIndexChanged.connect(
75     self.change_analysis_module)
76 self.combo_yvariable = QComboBox(panel)
77 self.combo_xvariable = QComboBox(panel)
78 self.combo_bandwidth_type = QComboBox(panel)
79 self.combo_bandwidth_type.addItems(["Rule of Thumb", "Constant"
80     , "Cross-Validation")
81 self.combo_bandwidth_type.currentIndexChanged.connect(self.
82     change_bandwidth_technique)
83
84 self.checkbox_custom_x_range = QCheckBox(panel)
85 self.checkbox_custom_x_range.stateChanged.connect(self.
86     on_custom_xrange_checkbox)
87 self.custom_x_range_minimum = QDoubleSpinBox(panel)
88 self.custom_x_range_minimum.setEnabled(False)
89 self.custom_x_range_maximum = QDoubleSpinBox(panel)
90 self.custom_x_range_maximum.setEnabled(False)
91
92 self.combo_priors = QComboBox(panel)
93 self.combo_priors.addItems(["One", "1/max value")
94 self.combo_priors.currentIndexChanged.connect(self.
95     change_prior_type)
96
97 self.bandwidth_spinner = QDoubleSpinBox(panel)
98 self.bandwidth_spinner.setValue(5.0)
99 self.bandwidth_spinner.setMinimum(-1.0)
100 self.bandwidth_spinner.setEnabled(False)
101 self.bin_spinner = QSpinBox(panel)
102 self.bin_spinner.setValue(20)
103 self.bin_spinner.setMaximum(999)
104
105 self.export_kde_button = QPushButton("Save KDE", panel)
106 self.export_kde_button.clicked.connect(self.save_results)
107 self.export_log_button = QPushButton("Save log regression", panel)
108 self.export_log_button.clicked.connect(self.save_log)
109
110 vlayout = QVBoxLayout(panel)
111 vlayout =QFormLayout(panel)
112 vlayout.addRow(self.open_directory)
113 vlayout.addRow("curve type: ", self.combo_curve_type)
114 vlayout.addRow("analysis module: ", self.
115     combo_analysis_module_type)
116 vlayout.addRow("y variable: ", self.combo_yvariable)
117 vlayout.addRow("x variable: ", self.combo_xvariable)
118 vlayout.addRow("bandwidth technique: ", self.
119     combo_bandwidth_type)
120 vlayout.addRow("bandwidth: ", self.bandwidth_spinner)
121 vlayout.addRow("number of y bins: ", self.bin_spinner)
122 vlayout.addRow(self.create_kde_button)
123 vlayout.addRow("custom x-range: ", self.checkbox_custom_x_range
124     )
125 vlayout.addRow("x-range min: ", self.custom_x_range_minimum)
126 vlayout.addRow("x-range max: ", self.custom_x_range_maximum)
127 vlayout.addRow("prior type: ", self.combo_priors)
128 vlayout.addRow(self.update_kde_button)
vlayout.addRow(self.export_kde_button)
vlayout.addRow(self.create_log_button)
vlayout.addRow(self.export_log_button)
#vlayout.addRow(self.create_gauss_button)
#vlayout.addRow("mask integral width: ", self.
→ mask_integral_width)
#vlayout.addRow("mask limit", self.mask_limit)
vlayout.addRow(self.create_mask_button)
vlayout.setAlignment(QtCore.Qt.AlignTop)
self.show()

def open_database(self):
    database_file, _ = QFileDialog.getOpenFileName(self,"open
→ database", ",";"database-file (*.ini)"
    if len(database_file) <= 0: return
self.open_database_from_file(database_file)

def open_database_from_file(self, filename):
    self.database = d.database(filename)
    self.combo_curve_type.clear()
    self.combo_curve_type.addItems(self.database.get_types())

def change_curve_type(self, index):
    curve_type = self.combo_curve_type.currentText()
    self.combo_analysis_module_type.clear()
    self.combo_analysis_module_type.addItems(self.database.
→ get_type_results(curve_type))

def on_custom_xrange_checkbox(self, state):
    if self.checkbox_custom_x_range.isChecked():
        self.custom_x_range_minimum.setEnabled(True)
        self.custom_x_range_maximum.setEnabled(True)
    else:
        self.custom_x_range_minimum.setEnabled(False)
        self.custom_x_range_maximum.setEnabled(False)

def change_prior_type(self, index):
    prior_type = self.combo_priors.currentText()
    if prior_type == 'One':
        self.kde.set_priors('One')
    if prior_type == '1/max value':
        print('maximum')
        self.kde.set_priors('Max')

def change_analysis_module(self, index):
    curve_type = self.combo_curve_type.currentText()
    analysis_module = self.combo_analysis_module_type.currentText()
    self.combo_xvariable.clear()
    self.combo_yvariable.clear()
    variables = self.database.get_result_files_columns(curve_type,
→ analysis_module)
    if variables is not None:
        self.combo_xvariable.addItems(variables)
        self.combo_yvariable.addItems(variables)

def change_bandwidth_technique(self, index):
    bandwidth_tech = self.combo_bandwidth_type.currentText()
if bandwidth_tech == 'Constant':
    self.bandwidth_spinner.setEnabled(True)
else:
    self.bandwidth_spinner.setEnabled(False)

def create_log(self):
    # creating a logistic regression
    self.loglearner = log.LogRegression()
    print("learning logistic regression")
    analysis_module = self.combo_analysis_module_type.currentText()
    # data
    X = np.array([], dtype=np.float64).reshape(2,0)
    # class info
    Y = np.array([], dtype=np.float64).reshape(1,0)
    curves = self.database.get_types()
    index = 0
    xname = self.combo_xvariable.currentText()
    yname = self.combo_yvariable.currentText()
    for curve_type in curves:
        for file in self.database.get_result_files(curve_type, analysis_module):
            # adding file to the data
            data, header, _ = d.load_data_asc(file)
            length = data[:, header.index(xname)].size
            t = np.vstack([data[:, header.index(xname)], data[:,
                header.index(yname)]]
            X = np.hstack([X, t])
            Y = np.hstack([Y, index*np.ones([1, length])])
            index += 1
        self.loglearner.add_data(np.transpose(X), np.transpose(Y))
        self.loglearner.xname = xname
        self.loglearner.yname = yname
        print("start learning")
        self.loglearner.learn()
        xmax = X[0,:].max()
        xmin = X[0,:].min()
        ymax = X[1,:].max()
        ymin = X[1,:].min()
        self.logrange = (xmin, xmax, ymin, ymax)
        print(self.logrange)
        self.plot_results(True)

def create_kde(self):
    print("learning")
    if self.database is None:
        return
    #
    curve_type = self.combo_curve_type.currentText()
    analysis_module = self.combo_analysis_module_type.currentText()
    yvar = self.combo_yvariable.currentText()
    mini, maxi = self.database.get_minmax_columns(curve_type,
        analysis_module, yvar)
    # setting the good bandwidth selection technique
    bandwidth_tech = self.combo_bandwidth_type.currentText()
    bandwidth_KDE_tech = None
if bandwidth_tech == 'Constant':
    bandwidth_KDE_tech = 'cns'
if bandwidth_tech == 'Cross-Validation':
    bandwidth_KDE_tech = 'cv'
if bandwidth_tech == 'Rule of Thumb':
    bandwidth_KDE_tech = 'rot'

self.kde = k.KDE_Learner(max_bin_value = maxi,
                         min_bin_value = mini,
                         bins = self.bin_spinner.value(),
                         bandwidth = self.bandwidth_spinner.
                         ↪ value(),
                         bandwidth_technique =
                         ↪ bandwidth_KDE_tech)

self.do_learning()

def create_gauss(self):
    print("learning")
    self.kde = gauss.GaussLearner(bins = self.bin_spinner.value())
    self.do_learning()

def do_learning(self):
    self.kde.xname = self.combo_xvariable.currentText()
    self.kde.yname = self.combo_yvariable.currentText()

    curve_type = self.combo_curve_type.currentText()
    analysis_module = self.combo_analysis_module_type.currentText()

    for file in self.database.get_result_files(curve_type,
                                             ↪ analysis_module):
        data, header, _ = d.load_data_asc(file)
        self.kde.add_data(data[:, header.index(self.kde.yname)],
                        data[:, header.index(self.kde.xname)])

        self.kde.learn()
        self.plot_results()

def create_mask(self):
    self.plot_results(True)

def plot_results(self, only_log = False):
    if not only_log:
        plot_range = self.kde.xrange
        if self.checkbox_custom_x_range.isChecked():
            plot_range[0] = self.custom_x_range_minimum.value()
            plot_range[1] = self.custom_x_range_maximum.value()
        self.plot_kde.full_clear()
        self.plot_kde.set_font_size(30)
        self.change_prior_type(0)
        x, y, z = self.kde.meshgrid(np.linspace(plot_range[0],
                                                ↪ plot_range[1],500)[:, np.newaxis])
        sc = self.plot_kde.axes().pcolor(x, y, z, cmap = 'jet' )
        cbar = self.plot_kde.plot.fig.colorbar(sc)
        cbar.set_label("prior\times probability density")
        xlabel = self.kde.xname
        ylabel = self.kde.yname
        if xlabel == 'force': xlabel = 'force (pN)'
        if ylabel == 'force': ylabel = 'force (pN)'

    self.plot_results()
if xlabel == 'l_p': xlabel = 'persistence length (nm)'
if xlabel == 'z': xlabel = 'relative extension'
if ylabel == 'z': ylabel = 'relative extension'
self.plot_kde.XY_label(xlabel,ylabel)
self.plot_kde.redraw()

if self.logrange is not None:
x, y, z = self.loglearner.meshgrid(self.logrange, index =
self.combo_curve_type.currentIndex())
print(x[0, :])
self.plot_mask.axes().pcolor(x, y, z, cmap = 'jet')
self.plot_mask.redraw()
# plot the mask
"""self.plot_mask.full_clear()
x, y, z = self.kde.mask(np.linspace(plot_range[0], plot_range
[1], 100)[:, np.newaxis],
  irange = self.mask_integral_width.value
  / 2)
  limit = self.mask_limit.value()
  self.plot_mask.axes().pcolor(x, y, (z > limit), cmap = 'Greys')
  self.plot_mask.XY_label(self.kde.xname, self.kde.yname)
  self.plot_mask.redraw()"

def save_results(self):
  """stores the results to a file""
  if self.database is None:
    return
  filename, _ = QFileDialog.getSaveFileName(self, "save results", ",KDE-result (*.tdms)"
  if len(filename) == 0:
    return
  if os.path.isfile(filename):
    dbase = database.KDEDatabase(filename)
  else:
    dbase = database.KDEDatabase()
  newKDE = database.KDE()
  newKDE.xvalues = np.linspace(self.kde.xrange[0], self.kde.xrange[1], 200)
  newKDE.xname = self.kde.xname
  newKDE.yname = self.kde.yname
  newKDE.name = self.combo_curve_type.currentText()
  newKDE.ybins = self.kde.get_bin_limits()
  newKDE.bandwidths = self.kde.bandwidths
  newKDE.version = self.kde.version()
  newKDE.priors = self.kde.priors
  newKDE.bandwidth_technique = self.combo_bandwidth_type.
  currentText()
  pdensity = self.kde.meshgrid(np.linspace(self.kde.xrange
  [0], self.kde.xrange[1], 200)[:, np.newaxis])
  newKDE.probability_density = pdensity.ravel()
  # set basic info
dbase.author = platform.node()
dbase.program_name = self.combo_analysis_module_type.
  currentText()
dbase.program_version = "0.1"
dbase.database_dictionary = self.database.get_info_dictionary()
dbase.add_kde(newKDE)
```python
dbase.write(filename)

## stores the logistic regression result to a file

def save_log(self):
    """stores the results to a file"""
    if self.database is None:
        return

    filename, _ = QFileDialog.getSaveFileName(self, "save results",
                                           "KDE-result (*.tdms)")
    if len(filename) == 0:
        return

    if os.path.isfile(filename):
        dbase = database.KDEDatabase(filename)
    else:
        dbase = database.KDEDatabase()

    newLog = database.LogReg()
    newLog.xname = self.loglearner.xname
    newLog.yname = self.loglearner.yname
    x, y, _ = self.loglearner.meshgrid(self.logrange, index = 0)
    newLog.xvalues = x[0, :]
    newLog.yvalues = y[:, 0]
    newLog.types = []
    newLog.probabilities = []

    for index in range(self.combo_curve_type.count()):
        newLog.types += [self.combo_curve_type.itemText(index)]
        _, _, v = self.loglearner.meshgrid(self.logrange, index =
                                            "")
        newLog.probabilities += [v.ravel()]

    dbase.log = newLog

    # set basic info
    dbase.author = platform.node()
    dbase.program_name = self.combo_analysis_module_type.
                           currentText()
    dbase.program_version = "0.1"
    dbase.database_dictionary = self.database.get_info_dictionary()
    dbase.write(filename)

if __name__ == '__main__':

    app = QApplication(sys.argv)
    ex = KDEWidget()
    sys.exit(app.exec_())
```

**Listing 10: binned_density_learner.py**

```python
# -*- coding: utf-8 -*-

"""this file contains a base class for estimating probability density
functions from data."

Created on Fri May 19 13:52:26 2017

@author: s115898
```

72
```python
import numpy as np

class BinnedDensityEstimator:
    def __init__(self, bins = 30, max_bin_value = 20, min_bin_value = 0):
        self.mbv = max_bin_value
        self.min_value = min_bin_value
        self.bin_range = self.mbv - self.min_value
        self.bin_count = bins
        self.bin_width = self.bin_range/self.bin_count
        self.xrange = [0, 1]
        self.xname = "l_p"
        self.yname = "force"
        self.data_list = []
        self.priors = np.ones(bins)
        for i in range(0, bins):
            self.data_list += [np.empty((0,1), float)]

    def get_bin(self, value = 0):
        # returns the index of the bin in which the value belong
        return np.floor((value - self.min_value)/self.bin_range * self.bin_count)

    def set_priors(self, technique):
        # set the values of the priors
        possible techniques are:
        'One': (all priors are 1),
        'Max': (priors are 1/maximum of pdf)"
        if technique == 'One':
            self.priors = np.ones(self.bin_count)
        elif technique == 'Max':
            for i in range(0,self.bin_count):
                max_value = self.get_bin_values(i,np.linspace(*self.xrange, 500)[:,np.newaxis]).max()
                if max_value > 0:
                    self.priors[i] = 1/max_value
                else:
                    self.priors[i] = 0

    def add_data_to_bin(self, bin_index, data):
        # adds data points to a bin"
        self.data_list[bin_index] = np.append(self.data_list[bin_index], data)

    def set_labels(self, label1, label2):
        # set labels for the x and y variable"
        self.xname = label1
        self.yname = label2

    def get_bin_values(self, bin_id, data_range):
        # returns the values of the pdf of a certain bin"
        pass
```

def get_bin_limits(self):
    limits = [self.min_value]
    for i in range(0, self.bin_count):
        limits += [self.min_value + (i+1)*self.bin_width]
    return np.array(limits)

def learn(self):
    """needs to be implemented by child class, this function gets called when all the data points are added and the pdf can be calculated""
    pass

def get_integral_range(self, index):
    pass

def meshgrid(self, data_range):
    """returns a meshgrid which can be plotted to visualize the pdfs""
    y_data = np.linspace(self.min_value, self.mbv, self.bin_count)
    x, y = np.meshgrid(data_range, y_data)
    z = np.zeros([y_data.size, data_range.size])
    for i in range(0, self.bin_count):
        z[i, :] = self.priors[i]*self.get_bin_values(i, data_range)
    return x, y, z

def mask(self, data_range, irange = 5):
    y_data = np.linspace(0, self.mbv, self.bin_count)
    x, y = np.meshgrid(data_range, y_data)
    z = np.zeros([y_data.size, data_range.size])
    for i in range(0, self.bin_count):
        for j in range(0, data_range.size):
            integral_range = self.get_integral_range(i)
            min_value = data_range[j]-integral_range
            max_value = data_range[j]+integral_range
            irange = np.linspace(min_value, max_value, 20)
            z[i, j] = np.trapz(self.get_bin_values(i, irange[:,np.newaxis]), irange)
    return x, y, z

def add_data(self, bin_values, data_points):
    #update minimum x-value
    if np.min(data_points) < self.xrange[0]:
        self.xrange[0] = np.min(data_points)
    #update maximum x-value
    if np.max(data_points) > self.xrange[1]:
        self.xrange[1] = np.max(data_points)
    #determine the bin in which each data point goes
    bins = self.get_bin(bin_values)
    for i in range(0, self.bin_count):
        self.add_data_to_bin(i, data_points[bins == i])

Listing 11: kde_learner.py
import numpy as np
import binned_density.learner as b
from sklearn.neighbors import KernelDensity
from sklearn.model_selection import GridSearchCV

class KDE_Learner(b.BinnedDensityEstimator):
    def __init__(self, bins=30, max_bin_value=20, min_bin_value=0, bandwidth=5, bandwidth_technique='cns'):
        b.BinnedDensityEstimator.__init__(self, bins, max_bin_value, min_bin_value)
        self.kde_list = [] #list of all KDE-estimators
        self.bandwidth = bandwidth
        self.bandwidth_technique = bandwidth_technique
        self.bandwidths = np.ones(bins)*bandwidth
        #self.xrange = [0, 1]
        for i in range(0, bins):
            if bandwidth > 0:
                self.kde_list += [KernelDensity(kernel='gaussian', bandwidth=bandwidth)]
            else:
                self.kde_list += [KernelDensity()]
            #self.data_list += [np.empty((0,1), float)]
    def version(self):
        return "0.1"

    def learn(self):
        if self.bandwidth_technique is "cv":
            params = {'bandwidth': np.linspace(1, 100, 20)}
            for i in range(0, self.bin_count):
                print("progress {:.1f}", i/self.bin_count*100)
                if self.data_list[i].size > 2:
                    params = {'bandwidth': np.logspace(-1, 1, 20)}
                    grid = GridSearchCV(KernelDensity(), params)
                    grid.fit(self.data_list[i].reshape(-1,1))
                    self.kde_list[i] = grid.best_estimator
                    self.bandwidths[i] = grid.best_estimator.bandwidth
            elif self.bandwidth_technique is 'cns':
                for i in range(0, self.bin_count):
                    print("progress {:.1f}", i/self.bin_count*100)
                    if self.data_list[i].size > 1:
                        self.kde_list[i].fit(self.data_list[i].reshape(-1,1))
            elif self.bandwidth_technique is 'rot':
                for i in range(0, self.bin_count):
                    print("progress {:.1f}", i/self.bin_count*100)
                    if self.data_list[i].size > 1:
                        A = np.std(self.data_list[i])
                        n = self.data_list[i].size
                        rot_bandwidth = 1.059 * A * n ** (-0.2)
if rot_bandwidth <= 0:
    rot_bandwidth = 1
self.bandwidths[i] = rot_bandwidth
self.kde_list[i] = KernelDensity(kernel='gaussian',
    bandwidth = rot_bandwidth)
self.kde_list[i].fit(self.data_list[i].reshape
    (-1,1))

def get_integral_range(self, index):
    return np.average(self.bandwidths)

def get_bin_values(self, bin_id, data_range):
    if self.data_list[bin_id].size > 2:
        return np.exp(self.kde_list[bin_id].score_samples(data_range))
    else:
        return np.zeros(data_range.size)

B.4 Utils

Listing 12: data_loading.py

# coding: utf-8

***
File containing function and classes to load result asc files
and getting information from the database
Created on Tue May 2 14:26:16 2017

@author: David den Boef
***
import numpy as np
import configparser as config
import glob as g
import os

def load_data_asc(filename = ''):
    """load the asc file""
    my_file = open(filename, 'r')
    file_info = my_file.readline()
    # check if the file can be read
    if not file_info.startswith("result_file version 0.1"):
        return None,None,None
    # number of lines of information
    infoline = []
    line_count = my_file.readline()
    for i in range(0, int(line_count)):
        info_line = my_file.readline();
        infoline +=[info_line[:-1]]
    # read the column headers
    line = my_file.readline()
    headers = line.split("\t")
    width = len(headers)-1
    # read the actual data
    return_array = np.zeros(width)
    for line in my_file:
        args = line.split("\t")
        temp = []
        for i in range(0,width):
temp += [float(args[i])]
return_array = np.vstack([return_array, np.array(temp)])
my_file.close()
return return_array[l:, :], headers[-1], infoline

def load_data_info(filename = ''):  
    """loads the metadata of the asc file, does not load all the data"""
    my_file = open(filename, 'r')
    file_info = my_file.readline()
    # check if the file can be read
    if not file_info.startswith("result_file version 0.1"):  
        return None, None  
    # number of lines of information
    infoline = []  
    line_count = my_file.readline()  
    for i in range(0, int(line_count)):  
        info_line = my_file.readline();  
        infoline += [info_line]  
    # read the column headers
    line = my_file.readline()  
    headers = line.split("\t")  
    my_file.close()
    return headers[:-1], infoline

class database:
    def __init__(self, filename):
        self.cfg = config.ConfigParser()
        self.cfg.read(filename)
        self.source_file = filename
        self.source_directory = os.path.dirname(os.path.abspath(filename))
    def get_types(self):
        """returns all the types of curves that are in the database"""
        return self.cfg.options("CurveDirectories")
    def get_type_results(self, curve_type):
        """returns a list of all analysis modules that have been performed on curve_type"""
        result_type = []  
        for x in os.walk(self.source_directory+'/'+self.cfg.get("CurveDirectories", curve_type)):  
            for dirname in x[1]:  
                if dirname.startswith("results_") and not dirname.
                    startswith("results_test."):  
                        if dirname[8:] not in result_type:
                            result_type.append(dirname[8:])
        return result_type
    def get_info_dictionary(self):
        """returns a dictionary describing the metadata of the database"""
        d = {}  
        for info in self.cfg.options("Info"):  
            d[info] = self.cfg.get("Info", info)
        for info in self.cfg.options("Test Set Info"):  
            d["test_set_"+info] = self.cfg.get("Test Set Info", info)
```python
    return d

def get_result_files(self, curve_type, analysis_type, only_first_file = False):
    """returns a list of all asc files that contain training results"""
    dir_ = self.source_directory+'/'+self.cfg.get("CurveDirectories", curve_type)
    files = []
    for file in g.glob(dir_+'/'+**/*.asc", recursive=True):
        if os.path.dirname(os.path.abspath(file)).endswith("results"+analysis_type):
            files += [file]
    if only_first_file:
        return [file];
    return files

def get_test_files(self, curve_type, analysis_type, only_first_file = False):
    """returns a list of all asc files that contain test results"""
    dir_ = self.source_directory+'/'+self.cfg.get("CurveDirectories", curve_type)
    files = []
    for file in g.glob(dir_+'/'+**/*.asc", recursive=True):
        if os.path.dirname(os.path.abspath(file)).endswith("results"+analysis_type):
            files += [file]
    if only_first_file:
        return files;
    return files

def get_test_performance(self, curve_type, analysis_type, threshold = 0.5):
    """returns a dictionary containing the performance info of a certain curve type"""
    files = self.get_test_files(curve_type, analysis_type)
    result_dictionary = {}
    for c in self.get_types():
        result_dictionary[c] = 0
        result_dictionary["failed"] = 0
    for f in files:
        data,headers,_ = load_data_asc(f)
        current_classification = "failed"
        classification_index = data.size
        classification_value = -1
        for c in self.get_types():
            #get the list of returned probilities for this curve
            probabilities = data[:, headers.index(c.replace(" ", "\s"))]
            index = np.where(probabilities > threshold)[0]
            #threshold has been exceeded
            if index.size > 0:
                #update classification if index is smaller than the current
                #classification index
                if classification_index > index[0]:
    ```
classification_index = index[0]
current_classification = c

# if the indexes are equal, set the classification to the one with the largest value
if classification_index == index[0] and
 probabilities[index[0]] >
 classification_value:
 classification_index = index[0]
classification_value = probabilities[index[0]]
current_classification = c

result_dictionary[current_classification] += 1
return result_dictionary

def get_performance_result(self, curve_type, analysis_type,
 threshold = 0.5):
 # return
data = {}

# first calculate the loss rate for this curve type
data = self.get_test_performance(curve_type, analysis_type,
 threshold)
wrong = 0
correct = 0
for key, value in data.items():
 if key == curve_type:
 correct += value
 else:
 wrong += value
loss_rate = wrong/(correct+wrong)
return_dictionary['loss'] = loss_rate

# next calculate what the fraction of other classes is that got
classified as this class
for c in self.get_types():
 if c != curve_type:
 return_dictionary[c] = np.zeros(threshold.size)

# same as get_performance_result but threshold is a numpy array
return_dictionary = {}
return_dictionary['loss'] = np.zeros(threshold.size)
for c in self.get_types():
 if c != curve_type:
 return_dictionary[c] = np.zeros(threshold.size)
for index in range(0, threshold.size):
    temp_dictionary = self.get_performance_result(curve_type, analysis_type, threshold[index])
    for key, value in temp_dictionary.items():
        return_dictionary[key][index] = value
return return_dictionary

def get_result_files_columns(self, curve_type, analysis_type):
    """Returns the name of the columns in the result files""
    files = self.get_result_files(curve_type, analysis_type, True)
    if len(files) > 0:
        print(files)
        return load_data_info(files[0])[0]
    else:
        return []

def get_minmax_columns(self, curve_type, analysis_type, column, testing = False):
    """Returns the minimum and the maximum of a certain column in the results""
    if not testing:
        files = self.get_result_files(curve_type, analysis_type)
    else:
        files = self.get_test_files(curve_type, analysis_type)
    minimum = None
    maximum = None
    mfile = None
    for file in files:
        data, header, info = load_data_asc(file)
        index = header.index(column)
        if minimum is None:
            minimum = data[0, index]
            maximum = data[0, index]
        minfile = data[:, index].min()
        maxfile = data[:, index].max()
        if minfile < minimum:
            minimum = minfile
            mfile = file
        if maxfile > maximum:
            maximum = maxfile
            #print(mfile)
    return minimum, maximum

def get_column_minimal_range(self, analysis_type, column, testing = False):
    minimum = None
    maximum = None
    for c in self.get_types():
        m1, m2 = self.get_minmax_columns(c, analysis_type, column, testing)
        #print("searched:"+c)
        #print((m1, m2))
        if minimum is None:
            minimum = m1
            maximum = m2
        if minimum < m1:
            minimum = m1
if maximum > m2:
    maximum = m2
return minimum, maximum

Listing 13: kde_database

# coding: utf-8

""
Created on Fri Jun 2 13:50:25 2017

@author: David den Boef
""
import numpy as np
from nptdms import TdmsWriter, RootObject, GroupObject, ChannelObject
from nptdms import TdmsFile
from scipy import interpolate
import data_loading as d

class KDE:
    """a class containing a single KDE""
    def __init__(self):
        self.xvalues = None
        self.ybins = None # bin limits
        self.probability_density = None # values of the pdf
        self.bandwidths = None # values of the bandwidth per bin
        self.name = None # name of the class / curve type
        self.xname = None # what is on the x axis
        self.yname = None # what is on the y axis
        self.bandwidth_technique = None
        self.version = None # version of the kde format
        self.priors = None #
        # interpolators
        self._interpolators = None

    def create_interpolators(self):
        """creates an interpolator for each bin""
        self._interpolators = []
        bin_count = self.ybins.size - 1
        width = self.xvalues.size
        for i in range(0, bin_count):
            data = self.probability_density[(i * width):(i * width + width)]
            self._interpolators += [interpolate.interp1d(self.xvalues, data)]

    def get_probability_density_value(self, yvalue, xvalue):
        """returns the value of the pdf at (xvalue, yvalue)""
        if type(xvalue) == float:
            index = np.argwhere(self.ybins > yvalue)
            if index.size == 0:
                return 0
            else:
                index = index[0][0]-1
                if index < 0:
                    return 0
if xvalue < self.xvalues[0]:
    return 0
if xvalue > self.xvalues[-1]:
    return 0
return self._interpolators[index](xvalue)

else:
    return_array = np.zeros(xvalue.size)
    for i in range(0,return_array.size):
        return_array[i] = self.get_probability_density_value(
            yvalue, float(xvalue[i]))
    return return_array

def get_value(self, **kwargs):
    yvalue = kwargs[self.yname]
    return self.get_prior(yvalue)*self.get_probability_density_value(yvalue, float(kwargs[self.xname]))

def get_average_x(self, yvalue):
    weight = np.zeros(self.xvalues.size)
    for index in range(0,self.xvalues.size):
        weight[index] = self.get_probability_density_value(yvalue, float(self.xvalues[index]))
    average = np.average(self.xvalues,weights = weight)
    average2 = np.average(self.xvalues**2,weights = weight)
    return average, np.sqrt(average2-average**2)

def get_bin_centers(self):
    centers = np.zeros(self.ybins.size-1)
    for i in range(0,self.ybins.size-1):
        centers[i] = (self.ybins[i]+self.ybins[i+1])/2
    return centers

def get_bandwidth(self, yvalue):
    """returns the bandwidth at yvalue""
    if self.ybins.size == 0:
        index = self.ybins.size - 2
        return 9999999
    else:
        index = np.argwhere(self.ybins > yvalue)
    if index.size == 0:
        return 9999999
    else:
        index = index[0][0]-1
    if index < 0:
        return 999999
    return self.bandwidths[index]

def get_prior(self, yvalue):
    """returns the prior at yvalue""
    if self.ybins.size == 0:
        index = self.ybins.size - 2
    else:
        index = np.argwhere(self.ybins > yvalue)
    if index.size == 0:
        index = self.ybins.size - 2
    else:
        index = index[0][0]-1
    if index < 0:
        index = 0
    return self.priors[index]

def _bin(self, yvalue):
    index = np.argwhere(self.ybins > yvalue)
    if index.size == 0:
        index = self.ybins.size - 2
    else:
        index = index[0][0] - 1
    if index < 0:
        index = 0
    return index

class LogReg:
    def __init__(self):
        self.xname = None
        self.yname = None
        self.types = []
        self.probabilities = []
        self.xvalues = None
        self.yvalues = None
        self._interpolators = []

def create_interpolators(self):
    self._interpolators = []
    import matplotlib.pyplot as plt
    for index in range(len(self.types)):
        height = self.yvalues.size
        width = self.xvalues.size
        p = np.zeros([height, width])
        for i in range(height):
            data = self.probabilities[index][(i * width):(i * width + width)]
            p[i, :] = data
        newInterpolator = interpolate.interp2d(self.xvalues, self.yvalues, p)
        self._interpolators += [newInterpolator]\n
def get_probabilities(self, **kwargs):
    return_value = np.zeros(len(self.types))
    for i in range(len(self.types)):
        return_value[i] = self._interpolators[i](kwargs[self.xname], kwargs[self.yname])
    return return_value

    def get_probabilities_ordered(self, ordering, **kwargs):
        return_values = np.zeros(len(self.types))
        for i in range(len(ordering)):
            ind = self.types.index(ordering[i])
            return_values[i] = self._interpolators[ind](kwargs['xname'], kwargs['yname'])
        return return_values

class Calibration:
    def __init__(self, filename):
        data, headers, info = d.load_data_asc(filename)
        self.bin_left = data[; headers.index('bin_left')]
        self.varname = info[0]
self.dictionary = {}
for i in range(len(headers)):
    if (headers[i] != 'bin_left') and (headers[i] != 'bin_right'):
        self.dictionary[headers[i]] = data[:, i]

def _bin(self, yvalue):
    index = np.argwhere(self.bin_left > yvalue)
    if index.size == 0:
        index = self.bin_left.size - 1
    else:
        index = index[0][0] - 1
    if index < 0:
        index = 0
    return index

def sigmoid(self, probability, slope, intercept):
    x = slope * probability + intercept
    return 1 / (1 + np.exp(-x))

def probabilities(self, ordering, probabilities, **kwargs):
    bin_index = self._bin(kwargs[self.varname])
    index = 0
    return_probabilities = np.zeros(probabilities.size)
    for name in ordering:
        coeff = self.dictionary['c' + name][bin_index]
        intercept = self.dictionary['i' + name][bin_index]
        return_probabilities[index] = self.sigmoid(probabilities[index], coeff, intercept)
        index += 1
    return return_probabilities / (np.sum(return_probabilities))

c = Calibration("../DataAnalysis/data/wlc_lock_lc_flip_v3.asc")
print(c.probabilities(['1 molecule', '2 molecules', 'no molecule'], np.array([0.4, 0.5, 0.6]), z = 0.3))

class KDEDatabase:
    """a class to manage a database of kdes stored in tdms format""
    def __init__(self, filename = None):
        self.kdes = []
        self.log = None
        self.weight = 1.0
        if filename is not None:
            self.tdms = TdmsFile(filename)
            self.author = self.info("Author")
            self.program_name = self.info("analysis module")
            self.program_version = self.info("analysis version")
            self.database_dictionary = self.tdms.object().properties
            del self.database_dictionary['Author']
            del self.database_dictionary['analysis module']
            del self.database_dictionary['analysis version']
            #adding all kde's to the list
            for kde in self.groups("KDE"):
                self.kdes += [self.get_kde(kde)]
            #load the logistic regression
            self.load_log()
else:
    self.tdms = None

def info(self, name):
    """returns the value of a root property with name""
    if self.tdms is None:
        return None
    return self.tdms.object().properties[name]

def add_kde(self, kde):
    """adds a kde to the database""
    self.kdes += [kde]

def load_log(self):
    if len(self.groups("LogReg")) == 0:
        return
    self.log = LogReg()
    self.log.xname = self.tdms.object("Logistic Regression").
        → property("xname")
    self.log.yname = self.tdms.object("Logistic Regression").
        → property("yname")
    for g in self.tdms.group_channels("Logistic Regression"):    
        channel = g.channel
        if (channel != self.log.xname) and (channel != self.log.
            → yname):
            self.log.types += [channel]
            self.log.probabilities += [g.data]
        elif channel == self.log.xname:
            self.log.xvalues = g.data
        elif channel == self.log.yname:
            self.log.yvalues = g.data
    self.log.create_interpolators()

def get_kde(self, name):
    """returns a kde class from the current tdms file""
    return_kde = KDE()
    return_kde.name = name
    return_kde.xname = self.tdms.object(name).property("xname")
    return_kde.yname = self.tdms.object(name).property("yname")
    return_kde.version = self.tdms.object(name).property("KDE
        → version")
    return_kde.bandwidth_technique = self.tdms.object(name).
        → property("bandwidth technique")
    return_kde.ybins = self.tdms.object(name,"bin-limits").data
    return_kde.xvalues = self.tdms.object(name,"x-axis").data
    return_kde.bandwidths = self.tdms.object(name,"bandwidth").data
    return_kde.probability_density = self.tdms.object(name,"
        → probability density").data
    return_kde.priors = self.tdms.object(name,"priors").data
    return_kde.create_interpolators()
    return return_kde

def groups(self, description_filter = None):
    """returns all the groups that are in the tdms file, use the
    → description filter
to filter out groups which don't have the same description""

85
if self.tdms is None:
    return None
if description_filter is None:
    return self.tdms.groups()
else:
    return_values = []
    for g in self.tdms.groups():
        if self.tdms.object(g).properties['Description'] == description_filter:
            return_values += [g]
    return return_values

def get_xaxis_name(self):
    return self.kdes[0].xname

def get_yaxis_name(self):
    return self.kdes[0].yname

def get_kde_value(self, name, **kwargs):
    for kde in self.kdes:
        if kde.name == name:
            return kde.get_value(**kwargs)
    return 0

def get_ordering(self):
    """returns the string that represent the ordering of the probabilities in the get_kde_probabilities function""
    if len(self.kdes) > 0:
        return_value = []
        for kde in self.kdes:
            return_value += [kde.name]
        return return_value
    elif self.log is not None:
        return self.log.types

def _kde(self, name):
    """returns the kde object by the given name""
    for kde in self.kdes:
        if kde.name == name:
            return kde
    return None

def get_kde_probabilities(self, **kwargs):
    """returns an array of probabilities per class for these values
    kwars is the dictionary that contains the name and values the
    parameters of the kde""
    if len(self.kdes) > 1:
        total = 0
        values = np.zeros(len(self.kdes))
        index = 0
        for kde in self.kdes:
            #values[index] = kde.get_prior(yvalue)*kde.get_probability_density(yvalue, xvalue)
            values[index] += self.get_kde_value(kde.name, **kwargs)
total += values[index]
index += 1
if total > 0:
    return values/total
else:
    """all possibilities are equally likely""
    return np.ones(len(self.kdes))/len(self.kdes)
elif len(self.kdes) == 1:
    return self.kdes[0].kde.get_value(**kwargs)
elif self.log is not None:
    return self.log.get_probabilities(**kwargs)
return None
def get_kde_probabilities_ordered(self, ordering, **kwargs):
    """returns an array of probabilities per class for these values
    ordering: an array containing the names of the classes, the
    output probabilities will be ordered in this way""
    if len(self.kdes) > 1:
        total = 0
        values = np.zeros(len(self.kdes))
        index = 0
        for name in ordering:
            #values[index] = kde.get_prior(yvalue)*kde.get_probability_density_value(yvalue, xvalue)
            values[index] += self.get_kde_value(name, **kwargs)
        total += values[index]
        index += 1
        if total > 0:
            return values/total
        else:
            return np.zeros(len(self.kdes))
elif len(self.kdes) == 1:
    return self.kdes[0].kde.get_value(**kwargs)
elif self.log is not None:
    return self.log.get_probabilities_ordered(ordering, **kwargs)
return None
def write(self, fname):
    #create the root properties
    root_properties = {"Author":self.author,
                        "analysis module":self.program_name,
                        "analysis version":self.program_version}
    root_properties = {...root_properties,**self.database_dictionary}
    if self.tdms is None:
        root_object = RootObject(properties=root_properties)
    else:
        root_object = self.tdms.object()
        root_object.properties = root_properties
        with TdmsWriter(fname) as tdms_writer:
            if self.tdms is not None:
                for g in self.groups("KDE"):
                    if self.should_copy(g):
                        self._copy_group(tdms_writer, g, root_object)
for k in self.kdes:
    self._write_kde(tdms_writer, k, root_object)
self._write_log(tdms_writer, root_object)

def _copy_group(self, tdms_writer, groupname, root):
group = self.tdms.object(groupname)
for c in self.tdms.group_channels(groupname):
    tdms_writer.write_segment([root, group, c])

def _should_copy(self, groupname):
    for n in self.kdes:
        if n.name == groupname:
            return False
    return True

def _write_kde(self, tdms_writer, kde, root):
    """write a kde to a tdms file""
    info_group = GroupObject(kde.name, properties={"Description": "KDE",
        "xname": kde.xname,
        "yname": kde.yname,
        "KDE version": kde.version,
        "bandwidth technique": kde.bandwidths})
xaxis = ChannelObject(kde.name, "x-axis", kde.xvalues)
ybins = ChannelObject(kde.name, "bin-limits", kde.ybins)
bandwidths = ChannelObject(kde.name, "bandwidth", kde.bandwidths)
priors = ChannelObject(kde.name, "priors", kde.priors)
probability_density = ChannelObject(kde.name, "probability density", kde.probability_density)

    tdms_writer.write_segment([root, info_group, xaxis])
    tdms_writer.write_segment([root, info_group, ybins])
    tdms_writer.write_segment([root, info_group, bandwidths])
    tdms_writer.write_segment([root, info_group, priors])
    tdms_writer.write_segment([root, info_group, probability_density])

##writes the logistic regression result to the tdms file
def _write_log(self, tdms_writer, root):
    if self.log is None:
        return
    info_group = GroupObject("Logistic Regression", properties={"Description": "LogReg",
        "xname": self.log.xname,
        "yname": self.log.yname})

    for n in self.kdes:
        if n.name == groupname:
            return False
    return True

def _write_kde(self, tdms_writer, kde, root):
    """write a kde to a tdms file""
    info_group = GroupObject(kde.name, properties={"Description": "KDE",
        "xname": kde.xname,
        "yname": kde.yname,
        "KDE version": kde.version,
        "bandwidth technique": kde.bandwidths})
xaxis = ChannelObject(kde.name, "x-axis", kde.xvalues)
ybins = ChannelObject(kde.name, "bin-limits", kde.ybins)
bandwidths = ChannelObject(kde.name, "bandwidth", kde.bandwidths)
priors = ChannelObject(kde.name, "priors", kde.priors)
probability_density = ChannelObject(kde.name, "probability density", kde.probability_density)

    tdms_writer.write_segment([root, info_group, xaxis])
    tdms_writer.write_segment([root, info_group, ybins])
    tdms_writer.write_segment([root, info_group, bandwidths])
    tdms_writer.write_segment([root, info_group, priors])
    tdms_writer.write_segment([root, info_group, probability_density])

##writes the logistic regression result to the tdms file
def _write_log(self, tdms_writer, root):
    if self.log is None:
        return
    info_group = GroupObject("Logistic Regression", properties={"Description": "LogReg",
        "xname": self.log.xname,
        "yname": self.log.yname})

    for n in self.kdes:
        if n.name == groupname:
            return False
    return True

def _write_kde(self, tdms_writer, kde, root):
    """write a kde to a tdms file""
    info_group = GroupObject(kde.name, properties={"Description": "KDE",
        "xname": kde.xname,
        "yname": kde.yname,
        "KDE version": kde.version,
        "bandwidth technique": kde.bandwidths})
xaxis = ChannelObject(kde.name, "x-axis", kde.xvalues)
ybins = ChannelObject(kde.name, "bin-limits", kde.ybins)
bandwidths = ChannelObject(kde.name, "bandwidth", kde.bandwidths)
priors = ChannelObject(kde.name, "priors", kde.priors)
probability_density = ChannelObject(kde.name, "probability density", kde.probability_density)

    tdms_writer.write_segment([root, info_group, xaxis])
    tdms_writer.write_segment([root, info_group, ybins])
    tdms_writer.write_segment([root, info_group, bandwidths])
    tdms_writer.write_segment([root, info_group, priors])
    tdms_writer.write_segment([root, info_group, probability_density])

##writes the logistic regression result to the tdms file
def _write_log(self, tdms_writer, root):
    if self.log is None:
        return
    info_group = GroupObject("Logistic Regression", properties={"Description": "LogReg",
        "xname": self.log.xname,
        "yname": self.log.yname})
for index in range(len(self.log.types)):
    probability_density = ChannelObject("Logistic Regression",
    self.log.types[index],
    self.log.probabilities[
        index])

    tdms.writer.write_segment([root, info_group,
        probability_density])

xaxis = ChannelObject("Logistic Regression", self.log.xname,
    self.log.xvalues)

yaxis = ChannelObject("Logistic Regression", self.log.yname,
    self.log.yvalues)

    tdms.writer.write_segment([root, info_group, xaxis])

    tdms.writer.write_segment([root, info_group, yaxis])

plt.pcolor(x, y, z)"

Listing 14: qtmatplotlib.py
```python
def set_font_size(size = 20):
    # set a font for the plots
    matplotlib.rcParams.update({'font.size': size})

set_font_size(30)

class mpl_canvas(FigureCanvas):
    """ a qwidget containing a matplotlib figure""

    def __init__(self, parent=None, width=5, height=4, dpi=100):
        self.fig = Figure(figsize=(width, height), dpi=dpi)
        self.axes = self.fig.add_subplot(111)

        FigureCanvas.__init__(self, self.fig)
        self.setParent(parent)

        FigureCanvas.setSizePolicy(self,
                                   QtWidgets.QSizePolicy.Expanding,
                                   QtWidgets.QSizePolicy.Expanding)
        FigureCanvas.updateGeometry(self)

    def get_axes(self):
        return self.axes

    def clear(self):
        self.fig.clear()
        self.axes = self.fig.add_subplot(111)

    def save(self, filename):
        self.fig.savefig(filename, bbox_inches='tight')

    def reset_axes(self, ax_info):
        self.fig.clear()
        self.axes = self.fig.add_axes(ax_info)

class mpl_widget(QtWidgets.QWidget):
    """ a qwidget containing a matplotlib window with a toolbar""

def __init__(self, parent = None):
    """creates the gui""
    QtWidgets.QWidget.__init__(self, parent)
    self.layout=QVBoxLayout(self)
    self.plot=mpl_canvas(self, width=5, height=4, dpi=100)
    self.toolbar=NavigationToolBar(self.plot, self)
    self.layout.addWidget(self.plot)
    self.layout.addWidget(self.toolbar)

    self.axes().tick_params(axis='y', direction='in')
    self.axes().tick_params(axis='x', direction='in')
    self.axes().tick_params(top = 'on')
```
def clear(self):
    """clear the plot""
    self.plot.get_axes().cla()

def full_clear(self):
    """clear the plot and also the axis information""
    self.plot.clear()
    #reset the style info
    self.axes().tick_params(axis='y', direction='in')
    self.axes().tick_params(axis='x', direction='in')
    self.axes().tick_params(top = 'on')
    self.axes().tick_params(right = 'on')

def XY_label(self, xlabel, ylabel):
    """set the xy labels of the plot""
    self.axes().set_xlabel(xlabel)
    self.axes().set_ylabel(ylabel)

def set_font_size(self, font_size):
    #set a font for the plots
    font = {'family' : 'serif',
            'weight' : 'normal',
            'size' : font_size}
    matplotlib.rc('font', **font)

def axes(self):
    """returns the axes so the user can plot""
    return self.plot.get_axes()

def redraw(self):
    """updates the plot""
    #check if the axes are locked. If this is the case
    #set the ax-limits the the locked values
    if self.axes_lock is not None:
        self.axes().set_xlim(*self.axes_lock[0])
        self.axes().set_ylim(*self.axes_lock[1])
    self.plot.draw()
    #self.plot.fig.tight_layout()
def lock_axes(self):
    """call this function to lock/unlock the axes"""
    if self.axes_lock is None:
        xlim = self.axes().get_xlim()
        ylim = self.axes().get_ylim()
        self.axes_lock = (xlim, ylim)
        self.lockview.setText("unlock view")
    else:
        self.axes_lock = None
        self.lockview.setText("lock view")