Figure 1: A raft of 500 fire ants, reproduced from Mlot et al.[3].

Python General purpose scripting.

SCons Build manager.

GNU Emacs Text editor.

Git Version control.

GNU/Linux/Gentoo Operating system and distribution.

and many, many more. I am deeply indebted to all of the smart, generous people who produce such wonderful tools. Besides providing the tools, they also provide very supportive communities to help haul complete newbies like me up the learning curve.

sawsim work was supported by National Institutes of Health Grants R01-GM071793.

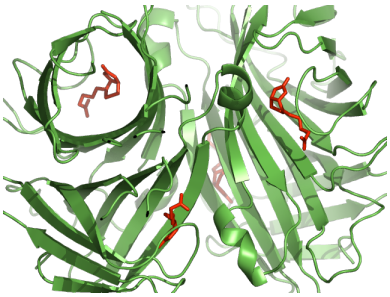


Figure 1.1: Complex of biotin (red) and a streptavidin tetramer (green) (PDB ID: 1SWE)[9]. The correct streptavidin conformation creates the biotin-specific binding pockets. Biotin-streptavidin is a model ligand-receptor pair isolated from the bacterium *Streptomyces avidinii*. Streptavidin binds to cell surfaces, and bound biotin increases streptavidin's cell-binding affinity[10]. Figure generated with PyMol.

1.2 Protein folding energy landscapes

Finding a protein's lowest energy state via a brute force sampling of all possible conformations is impossibly inefficient, due to the exponential scaling of possible conformations with protein length, as outlined by Levinthal[37]. This has lead to a succession of models explaining the folding mechanism. For a number of years, the "pathway" model of protein folding enjoyed popularity (Fig. 1.2a)[37]. More recently, the "landscape" or "funnel" model has come to the fore (Fig. 1.2b)[13]. Both of these models reduce the conformation space to a more approachable analog, and their success depends on striking a useful balance between simplicity and accuracy.

When the choice of theoretical approach becomes murky, you must gather experimental data to help distinguish between similar models. Separating the pathway model from the funnel model is only marginally within the realm of current experimental techniques, but with higher throughput and increased automation it should be easier to make such distinctions in the near future.

1.3 Why *single* molecule?

The large size of proteins relative to simpler molecules limits the information attainable from bulk measurements, because the macromolecules in a population can have diverse conformations and behaviors. Bulk measurements average over these differences, producing excellent statistics for the mean, but making it difficult to understand the variation. The individualized, and sometimes rare,

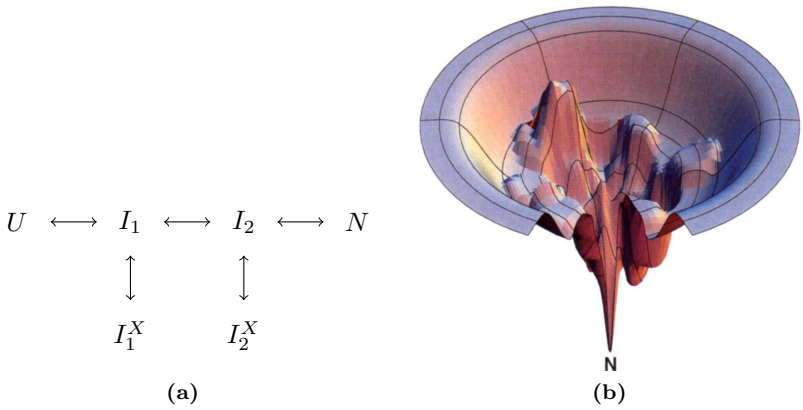


Figure 1.2: (a) A "double T" example of the pathway model of protein folding, in which the protein proceeds from the native state N to the unfolded state U via a series of metastable transition states I_1 and I_2 with two "dead end" states I_1^X and I_2^X. Adapted from Bédard et al.[12]. (b) The landscape model of protein folding, in which the protein diffuses through a multi-dimensional free energy landscape. Separate folding attempts may take many distinct routes through this landscape on the way to the folded state. Reproduced from Dill and Chan[13].

behaviors of macromolecules can have important implications for their functions inside the cell. Single molecule techniques, in which the macromolecules are studied one at a time, allow direct access to the variation within the population without averaging. This provides important and complementary information about the functional mechanisms of several biological systems[38].

Single molecule techniques provide an opportunity to study protein folding and unfolding at the level of a single molecule, where the distinction between the pathway model and funnel model is clearer. They also provide a convenient benchmark for verifying molecular dynamics simulations, because it takes lots of computing power to simulate even one biopolymer with anything close to atomic resolution over experimental time scales. Even with significant computing resources, comparing molecular dynamics results with experimental data remains elusive. For example, experimental pulling speeds are on the order of μm/s, while simulation pulling speeds are on the order of m/s[39–43].

Single molecule techniques for manipulating biopolymers include optical measurements, *i.e.*, single molecule fluorescence microscopy and spectroscopy, and mechanical manipulations of individual macromolecules, *i.e.*, force microscopy and spectroscopy using atomic force microscopes (AFMs), laser tweezers[44,45], magnetic tweezers[46], biomembrane force probes[47], and centrifugal mi-

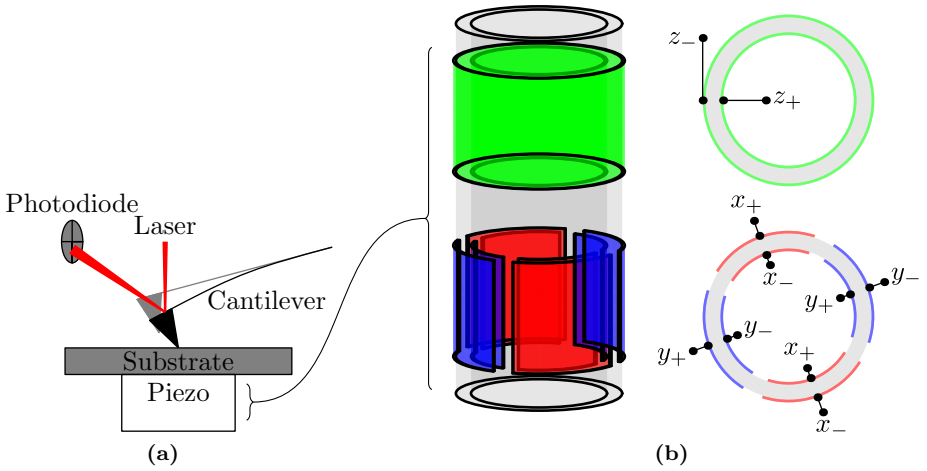


Figure 2.1: (a) Operating principle for an Atomic Force Microscope. A sharp tip integrated at the end of a cantilever interacts with the sample. Cantilever bending is measured by a laser reflected off the cantilever and incident on a position sensitive photodetector. (b) Schematic of a tubular piezoelectric actuator. In our AFM, the substrate is mounted on the top end of the tube, and the bottom end is fixed to the microscope body. This allows the piezo to control the relative position between the substrate and the AFM cantilever. The electrodes are placed so radial electric fields can be easily generated. These radial fields will cause the piezo to expand or contract axially. The z voltage causes the tube to expand and contract uniformly in the axial direction. The x and y voltages cause expansion on one side of the tube, and contraction (because of the reversed polarity) on the other side of the tube. This tilts the tube, shifting the sample horizontally.

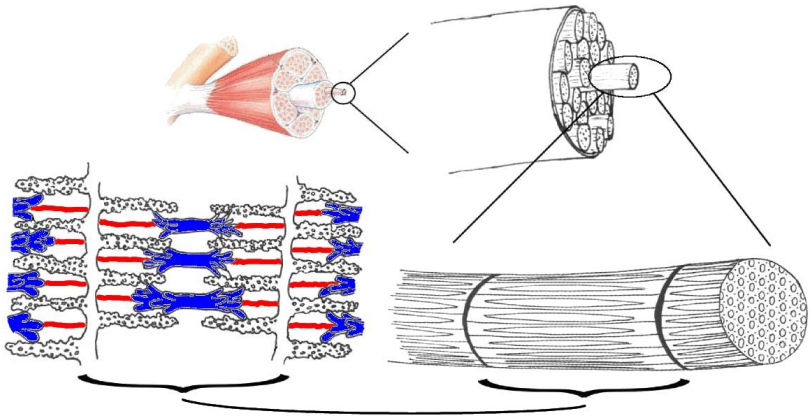


Figure 2.2: Biological role of titin. Moving clockwise from the upper left you can see a bone/muscle group, a muscle fiber, a myofibril, and a sarcomere. In the sarcomere, the white, knobbly filaments are actin. The myosin bundles are blue, and the titin linkers are red. When the muscle contracts, the myosin heads walk up the actin filiaments, shortening the sarcomere. When the muscle relaxes, the myosin heads release the actin filimants and slide back, lengthening the sarcomere. Titin functions as an entropic spring that keeps the myosin from falling out of place during the passive, relaxed stage. This figure is adapted from Wikipedia [14].

hundreds of nanonewtons. The investigation of the unfolding and refolding processes of individual protein molecules by the AFM is feasible because many globular proteins unfold under external forces in this range. Since elucidating the mechanism of protein folding is currently one of the most important problems in biological sciences, the potential of the AFM for revealing significant and unique information about protein folding has stimulated much effort in both experimental and theoretical research.

2.2 Protein polymer synthesis—Titin I27

Early experiments in force spectroscopy involved DNA [27,57], but before long they were also investigating proteins. Native titin was one of the first proteins studied with force spectroscopy [30]. Titin is a muscle protein involved in passive elasticity (Fig. 2.2), so it is an ideal subject when examining the effect of mechanical force [58]. Titin is also interesting because, while it is one of the largest known proteins, it is composed of a series of globular domains. When Rief et al. [30] carried out their seminal unfolding experiment, they observed a very characteristic sawtooth as the domains unfolded (see Section 2.4 for a discussion of these sawteeth).

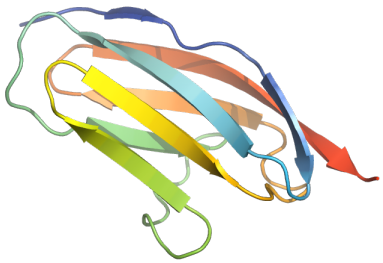


Figure 2.3: I27, the immunoglobulin-like domain 27 from human titin (PDB ID: 1TIT)[15]. The entire domain is 4.7 nm from end to end. Figure generated with PyMol.

Unfortunately, it is difficult to analyze the unfolding of native titin, because the heterogenous globular domains make it hard to attribute a particular subdomain to a partuclar unfolding event. Unfolding a single domain is not feasable because the large radius of curvature of an AFM tip ($\sim$ 20 nm[59]) dwarfs the radius of a globular domain ($\sim$ 2 nm[15]). When such a large tip is so close to the substrate, van der Waals forces and non-specific binding with the surface dominate the tip-surface interaction. In order to increase the tip-surface distance while preserving single molecule analysis, Carrion-Vazquez et al.[6] synthesized a protein composed of eight repeats of immunoglobulin-like domain 27 (I27), one of the globular domains from native titin (Fig. 2.3). Octameric I27 produced using their procedure is now available commercially[60].

Synthetic proteins are generally produced by creating a plasmid coding for the target protein, inserting the plasmid in a bacteria, waiting while the bacteria produce your protein, and then purifying your proteins from the resulting culture. In this case, Carrion-Vazquez et al.[6] extracted messenger RNA coding for titin from human cardiac tissue[30], and used reverse transcriptase to generate a complementary DNA (cDNA) library from human cardiac muscle messenger RNA. This cDNA is then amplified using the polymerase chain reaction (PCR), with special primers that allow you to splice the resulting cDNA into a plasmid (which ends up with one I27). Then they ran another PCR on the plasmid, linearized the plasmid with two restriction enzymes, and grafted two I27-containing sections together to form a new plasmid (now with two I27s, Fig. 2.4). Another PCR-split-join cycle produced a plasmid with four I27s, and a final cycle produced a plasmid with eight. The eventual plasmid vector has the eight I27s and a host-specific promoter that causes the

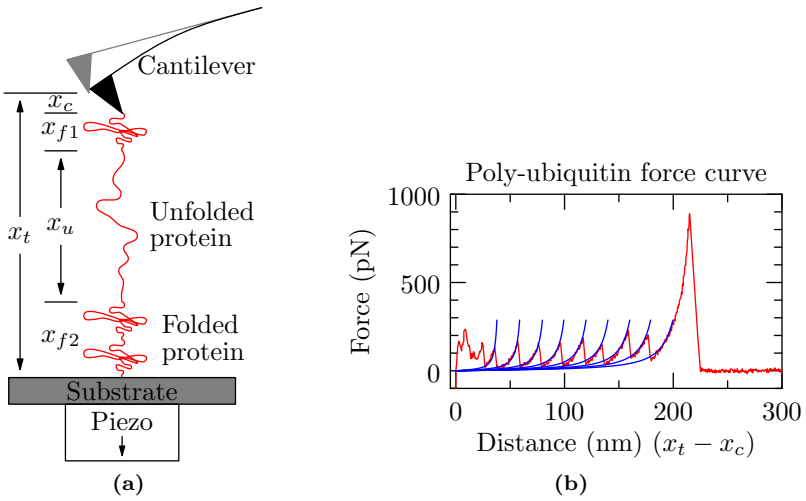


Figure 2.5: (a) Schematic of the experimental setup for mechanical unfolding of proteins using an AFM (not to scale). An experiment starts with the tip in contact with the substrate surface, which is then moved away from the tip at a constant speed. x_t is the distance traveled by the substrate, x_c is the cantilever deflection, x_u is the extension of the unfolded polymer, and $x_f = x_{f1} + x_{f2}$ is the extension of the folded polymer. (b) An experimental force curve from stretching a ubiquitin polymer (—) with the rising parts of the peaks fitted to the WLC model (—, Section 3.2.1)[17]. The pulling speed used was 1 μm/s. The irregular features at the beginning of the curve are due to nonspecific interactions between the tip and the substrate surface, and the last high force peak is caused by the detachment of the polymer from the tip or the substrate surface. Note that the abscissa is the extension of the protein chain $x_t - x_c$.

its ends, each protein molecule feels the externally applied force, which increases the probability of unfolding by reducing the free energy barrier between the native and unfolded states. The unfolding of one molecule in the polymer causes a sudden lengthening of the polymer chain, which reduces the force on each protein molecule and prevents another unfolding event from occurring immediately. The force versus extension relationship, or *force curve*, shows a typical sawtooth pattern (Fig. 2.5b), where each peak corresponds to the unfolding of a single protein domain in the polymer. Therefore, the individual unfolding events are separated from each other in space and time, allowing single molecule resolution despite the use of multi-domain test proteins.

2.5 Cantilever spring constant calibration

In order to measure forces accurately with an AFM, it is important to measure the cantilever spring constant κ. The force exerted on the cantilever can then be deduced from its deflection via Hooke's

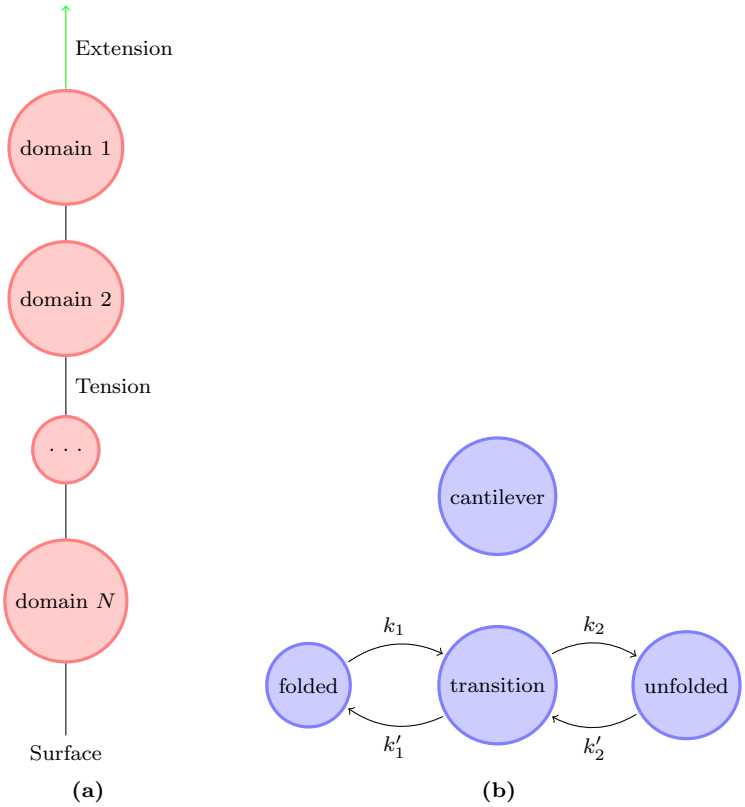


Figure 3.1: (a) Extending a chain of domains. One end of the chain is fixed, while the other is extended at a constant speed. The domains are coupled with rigid linkers, so the domains themselves must stretch to accomodate the extension. Compare with Fig. 2.5a. (b) Each domain exists in a discrete state. At each timestep, it may transition into another state following a user-defined state matrix such as this one, showing a metastable transition state and an explicit "cantilever" domain.

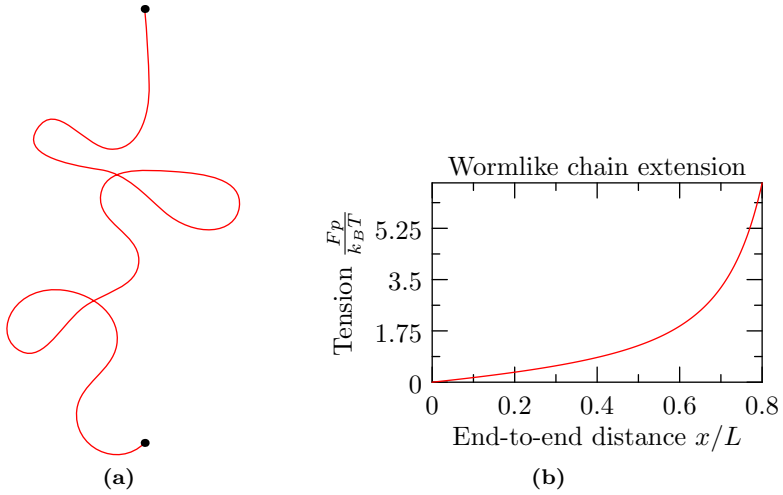


Figure 3.2: (a) The wormlike chain models a polymer as an elastic rod with persistence length p and contour length L. (b) Force vs. extension for a WLC using Bustamante's interpolation formula.

around 11 pN. Most proteins studied using force spectroscopy have unfolding forces in the hundreds of piconewtons, by which point the interpolation formula is in it's more accurate high-extension regime.

For chain with N_u unfolded domains sharing a persistence length p_u and per-domain contour lengths L_{u1}, the tension of the WLC is determine by summing the contour lengths

$$F(x, p_u, L_u, N_u) = F_{\mathrm{WLC}}(x, p_u, N_u L_{u1}) \; . \tag{3.5}$$

Folded domains

Short chains of folded proteins, however, are not easily described by polymer models. Several studies have used WLC and FJC models to fit the elastic properties of the modular protein titin[98,99], but native titin contains hundreds of folded and unfolded domains. For the short protein polymers common in mechanical unfolding experiments (Section 2.2), the cantilever dominates the elasticity of the polymer-cantilever system before any protein molecules unfold. After the first unfolding event occurs, the unfolded portion of the chain is already longer and softer than the sum of all the remaining folded domains, and dominates the elasticity of the whole chain. Therefore, the details

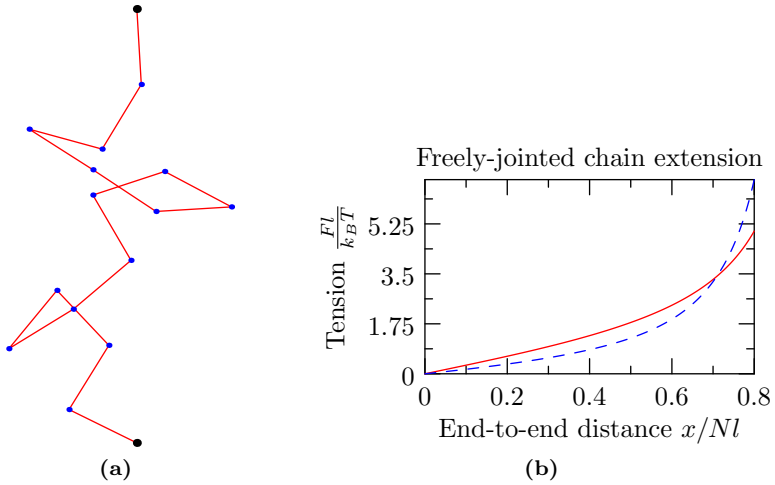


Figure 3.3: (a) The freely-jointed chain models the polymer as a series of N rigid links, each of length l, which are free to rotate about their joints. Each polymer state is a random walk, and the density of states for a given end-to-end distance is determined by the number of random walks that have such an end-to-end distance. (b) Force vs. extension for a hundred-segment FJC. The WLC extension curve (with $p = l$) is shown as a dashed line for comparison.

tether point and the position of the tip (Eq. (3.2) is scalar, not vector, addition). The effects of this assumption are also minimized due to greater length of the unfolded domain compared with the other domains (folded proteins and cantilever deflection). For example, a 0.050 N/m cantilever under 200 pN of tension will bend $x_c = F/\kappa_c = 4$ nm. The entire end-to-end length of folded domains such as I27 are also around 5 nm (Fig. 2.3). A single unfolded I27, with its 89 amino acids[15], should have an unfolded contour length of 89 aa $\cdot$ 0.38 nm = 33.8 nm, equivalent to a cantilever and five folded domains.

Velocity-clamp example

Consider an experiment pulling a polymer with N identical protein domains at a constant speed. At the start of an experiment, the chain is unstretched ($x_t = 0$), which means all the domains are unstretched, the cantilever is undeflected, and the tip is in contact with the surface. There is one domain in the cantilever state, N in the folded state, and none in the unfolded state. As the surface moves away from the tip at a constant speed v, the chain becomes more extended (Fig. 2.5a), such

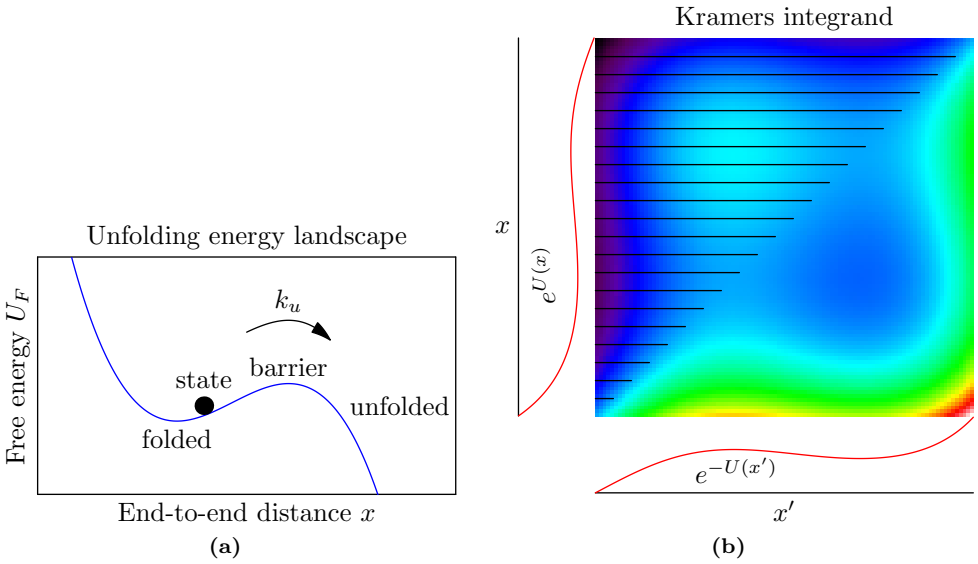


Figure 3.6: (a) Energy landscape schematic for Kramers integration (compare with Fig. 3.4). (b) A map of the magnitude of Kramers' integrand, with black lines tracing the integration region. The bulk of the contribution to the integral comes from the bump in the upper left, with x near the boundary and x' near the folded state. This is why you can calculate a close approximation to this integral by restricting the integration to $x_{\min}$ and $x_{\max}$, located a few k_BT beyond the folded and transition states respectively. The restricted integral is much easier to calculate numerically than one bound by $\pm\infty$. (Eq. (3.12)).

sense for sufficiently sharp folded and transition states, where these two measurements will capture the shape of the large-integrand region (Fig. 3.6b). The steepest-descent formulation has less to say about the underlying energy landscape, but it may be more robust in the face of noisy data.

How to choose which unfolding model to use? For proteins with relatively narrow folded and transition states, the Bell model provides a good approximation, and it is the model used by the vast majority of earlier work in the field. I will use the Bell model in my analysis of ion-dependent unfolding (Chapter 7), but analyzing my unfolding data with a different transition rate model is just a matter of changing some command line options and rerunning the sawsim simulations.

Assumptions

The interactions between different parts of the polymer and between the chain and the surface (except at the tethering points) are often ignored. This is usually reasonable since these interactions should not make substantial contributions to the force curve at the force levels of interest, where

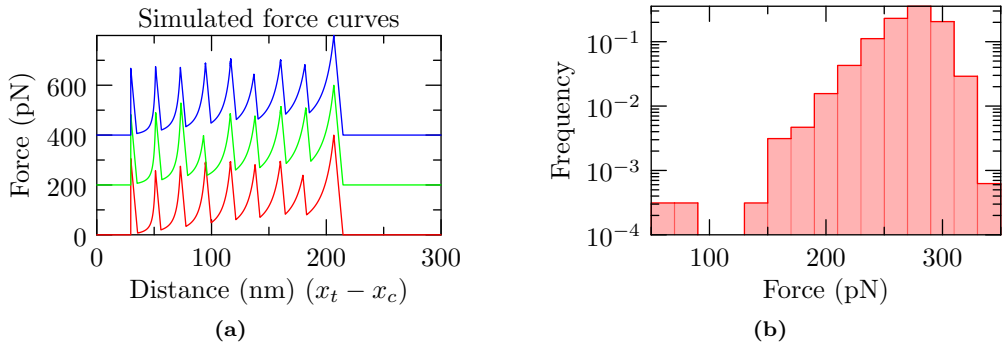


Figure 3.7: (a) Three simulated force curves from pulling a polymer of eight identical protein molecules. The simulation was carried out using the parameters: pulling speed $v = 1$ μm/s, cantilever spring constant $\kappa_c = 50$ pN/nm, temperature $T = 300$ K, persistence length of unfolded proteins $p_u = 0.40$ nm, $\Delta x_u = 0.225$ nm, and $k_{u0} = 5 \cdot 10^{-5}$ s^{-1}. The contour length between the two linking points on a protein molecule is $L_{f1} = 3.7$ nm in the folded form and $L_{u1} = 28.1$ nm in the unfolded form. These parameters are those of ubiquitin molecules connected through the N-C termini[17,19]. Detachment from the tip or substrate is assumed to occur at a force of 400 pN. In experiments, detachments have been observed to occur at a variety of forces. For clarity, the green and blue curves are offset by 200 and 400 pN respectively. (b) The distribution of the unfolding forces from 400 simulated force curves (3200 data points) such as those shown in (a). The frequency is normalized by the total number of points, *i.e.*, the height of each bin is equal to the number of data points in that bin divided by the total number of data points.

Because the unfolding behaviors of an individual sawtooth curve is stochastic (Fig. 3.7a), we cannot directly compare single curves in our fit quality metric. Instead, we gather many experimental and simulated curves, and compare the aggregate properties. For velocity-clamp experiments, the usual aggregate property used for comparison is a histogram of unfolding forces[6] (Fig. 3.7b). Defining and extracting "unfolding force" is suprisingly complicated (Section 6.3), but basically it is the highest tension force achieved by the chain before an unfolding event (the drops in the sawtooth). The final drop is not an unfolding event, it is the entire chain breaking away from the cantilever tip, severing the connection between the substrate and the cantilever.

3.3.2 The supramolecular scaffold

Analysis of the mechanical unfolding data is complicated by the dependence of the average unfolding force on the unfolding order due to the serial linkage of the molecules. Under an external stretching force F, the probability of some domain unfolding in a polymer with N_f folded domains is $N_f P_1$

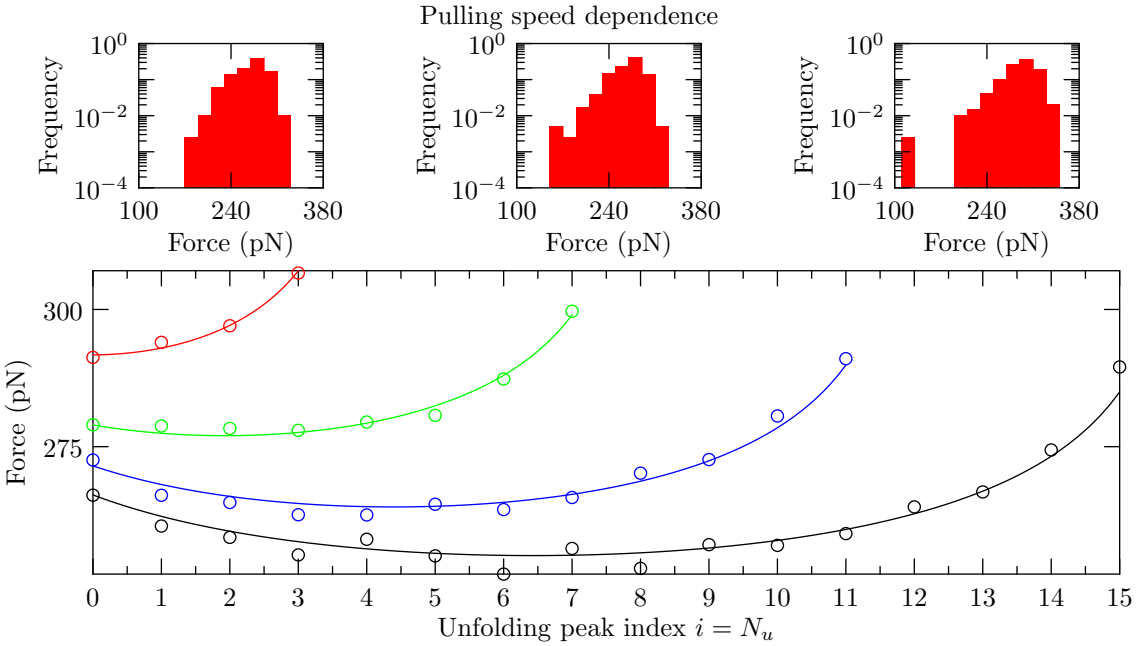


Figure 3.8: The dependence of the unfolding force on the temporal unfolding order for four polymers with 4, 8, 12, and 16 identical protein domains. Each point in the figure is the average of 400 data points. The first point in each curve represents the average of only the first peak in each of the 400 simulated force curves, the second point represents the average of only the second peak, and so on. The solid lines are fits of Eq. (3.21) to the simulated data, with best fit $\kappa_{\mathrm{WLC}} = 203$, 207, 161, and 157 pN/nm, respectively, for lengths 4 through 16. The insets show the force distributions of the first, fourth, and eighth peaks, left to right, for the polymer with eight protein domains. The parameters used for generating the data were the same as those used for Fig. 3.7a, except for the number of domains. The histogram insets were normalized in the same way as in Fig. 3.7b.

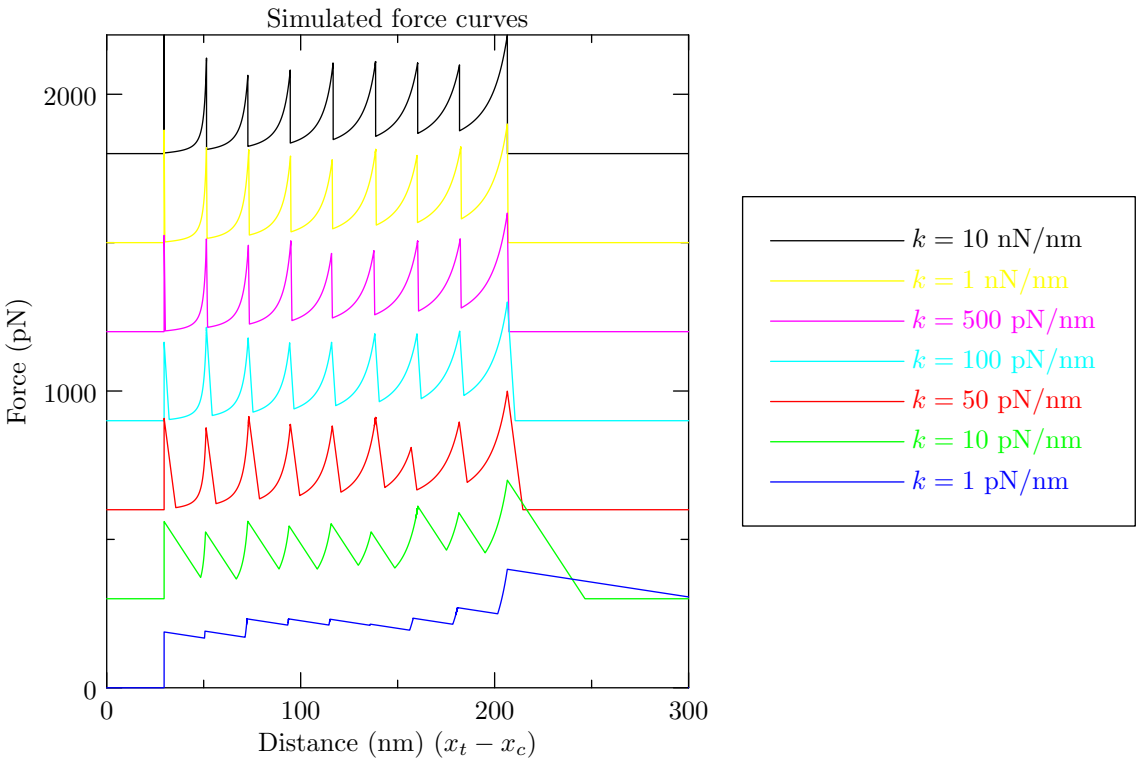


Figure 3.9: Simulated force curves obtained from pulling a polymer with eight protein molecules using cantilevers with different force constants κ_c. Parameters used in generating these curves are the same as those used in Fig. 3.7, except the cantilever force constant. Successive force curves are offset by 300 pN for clarity.

is small in comparison with the contour length increment from the unfolding of a single molecule. Figure 3.9 also shows that the back side of the force peaks becomes more tilted as the cantilever becomes softer. This is due to the fact that the extension (end-to-end distance) of the protein polymer has a large sudden increase as the tension rebalances after an unfolding event.

It should also be mentioned that the contour length increment from each unfolding event is not equal to the distance between adjacent peaks in the force curve because the chain is never fully stretched. This contour length increase can only be obtained by fitting the curve to WLC or other polymer models (Fig. 2.5b).

3.3.4 Determination of Δx_u and k_{u0}

As mentioned in Section 3.3.1, fitting experimental unfolding force histograms to simulated histograms allows you to extract best-fit parameters for your simulation model. For example, if you

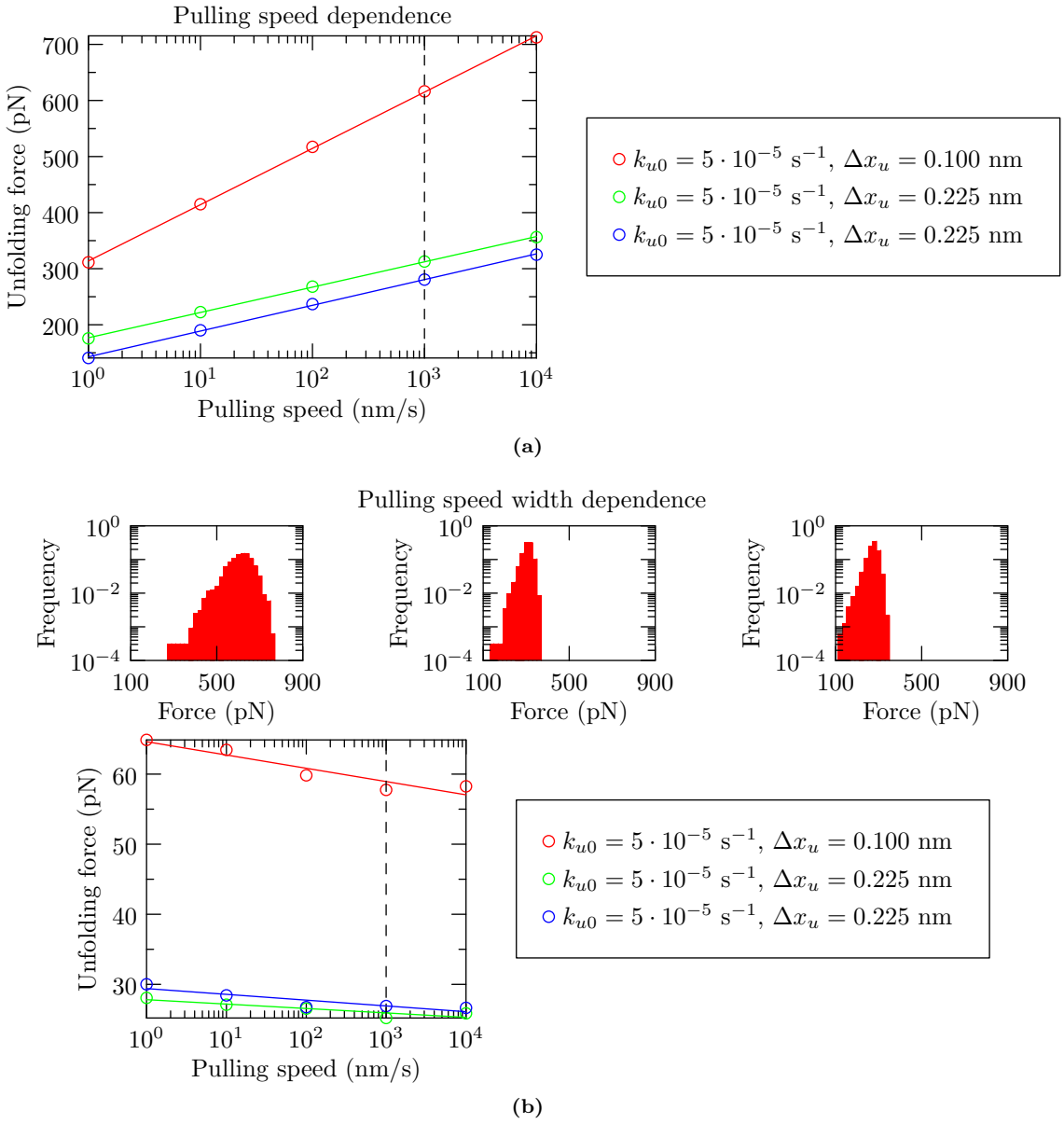


Figure 3.10: (a) The dependence of the unfolding forces on the pulling speed for three different model protein molecules characterized by the parameters k_{u0} and Δx_u. The polymer length is eight molecules, and each symbol is the average of 3200 data points. (b) The dependence of standard deviation of the unfolding force distribution on the pulling speed for the simulation data shown in (a), using the same symbols. The insets show the force distribution histograms for the three proteins at the pulling speed of 1 μm/s. The left, middle and right histograms are for the proteins represented by the top, middle, and bottom lines in (a), respectively.

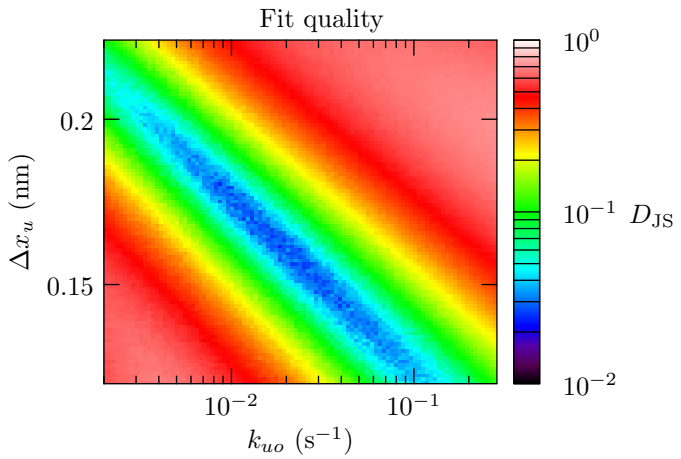


Figure 3.11: Fit quality between an experimental data set and simulated data sets obtained using various values of unfolding rate parameters k_{u0} and Δx_u. The experimental data are from octameric ubiquitin pulled at 1 μm/s[17], and the other model parameters are the same as those in Fig. 3.7. The best fit parameters are $\Delta x_u = 0.17$ nm and $k_{u0} = 1.2 \cdot 10^{-2}$ s^{-1}. The simulation histograms were built from 400 pulls at for each parameter pair.

```
3.2e-10 1

#HISTOGRAM: -v 8e-7
#Force (N)      Unfolding events
1.4e-10 0
1.5e-10 3
...
3.2e-10 50
3.3e-10 13

#HISTOGRAM: -v 1e-6
#Force (N)      Unfolding events
1.5e-10 2
1.6e-10 3
...
3.3e-10 24
3.4e-10 2
```

Each sawsim run simulates a single sawtooth curve, so you need to run many sawsim instances to generate your simulated histograms. To automate this task, sawsim comes with a Python wrapping library (`pysawsim`), which provides convenient programmatic and command line interfaces for generating and manipulating sawsim runs. For example, to compare the experimental histograms listed above with simulated data over a 50-by-50 grid of k_{u0} and Δx, you would use something like

```
$ sawsim_hist_scan.py -f '-s cantilever,hooke,0.05 -N1 -s folded,null -N8
>   -s "unfolded,wlc,{0.39e-9,28e-9}" -k "folded,unfolded,bell,{%g,x%g}"
>   -q folded' -r '[1e-5,1e-3,50],[0.1e-9,1e-9,50]' --logx histograms.txt
```

That's a bit of a mouthful, so let's break it down. Without the sawsim template (`-f ...`), we can focus on the comparison options:

```
$ sawsim_hist_scan.py ... -r '[1e-5,1e-3,50],[0.1e-9,1e-9,50]' --logx histograms.txt
```

This sets up a two-parameter sweep, with the first parameter going from $1 \cdot 10^{-5}$ to $1 \cdot 10^{-3}$ in 50 logarithmic steps, and the second going from $0.1 \cdot 10^{-9}$ to $1 \cdot 10^{-9}$ in 50 linear steps. The sawsim template defines the simulation model (Fig. 3.1 and Table 3.1), and `%g` marks the location where the swept parameters will be inserted.

Behind the scenes, `pysawsim` is spawning several concurrent sawsim processes to take advantage of any parallel processing facilities you may have access to (e.g. multiple cores, MPI, PBS, ...). A 50-by-50 grid with 400 runs per pixel at about one second per sawsim pull would take arount 12 days of serial execution. Moving the simulation to the departments' 16 core file server cuts that execution time down to 18 hours, which will easily complete over a quiet weekend. Using MPI on the departments' 15 box, dual core computer lab, the simulation would finish overnight.

3.3.6 Testing

Once a body of code reaches a certain level of complication, it becomes difficult to convince others (or yourself) that it's actually working correctly. In order to test sawsim, I've developed a test suite (distributed with sawsim) that compares simulated unfolding force histograms with analytical histograms for a number of situations where solving for the analytical histogram is possible. In the following subsection, I'll work out the theoretical unfolding force distribution for a number of tractable cases. The sawsim test suite generates simulated unfolding curves for these tractable cases (e.g. single domain Bell model unfolding with a constant loading rate), and compares the simulated unfolding force histograms with the expected theoretical distribution. The simulated histograms match the theoretical distributions for each combination of models regardless of the parameters you

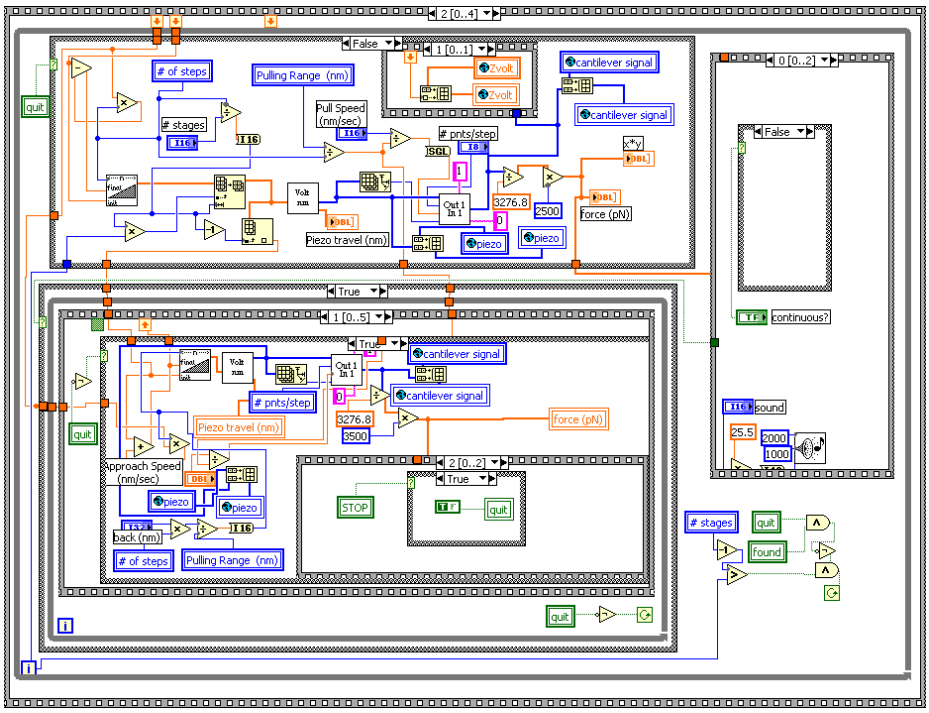


Figure 4.1: An excerpt from the main frame of the LabVIEW stack. This frame codes for the velocity-clamped pull phase of a push–bind–pull experiment.

subroutines in *virtual instruments* (VIs).

The problem comes when you want to update one of your subroutines. LabVIEW VIs are linked dynamically by VI name[132], so there was no easy way to swap a new version of the VI into your application for testing without renaming the subroutine. With the Project Explorer (new in LabVIEW 8.0[132], released 2005), these renames became easier. However, throughout my time in the Yang lab, the Windows machines all ran LabVIEW 7.1 (released in 2004).

Because of difficulties with name-based VI linking and the relative inexperience of many scientists in the maintenance benefits of modular programming[133,134], LabVIEW code often ends up without a clean separation between high-level and low-level tasks (Fig. 4.1). This lack of structure makes it difficult to reuse existing code to address similar tasks.

The second obstacle to maintaining LabVIEW code is the binary file format for VIs. The established method for recording software history is to use a version control system (VCS), which records versions of the project in a repository. Each change to the project is committed to the

```
static int set_digital_output_data(DIGITAL_OUTPUT *d, unsigned int data)
{
  d->data = (uInt32) data;
  DAQmxErrChk_struct( Write_WriteDigPort(d->taskHandle, d->data) );
 Error:
  if (d->error != 0) {
    CHK( close_digital_output(d) );
    M_EXIT(FAILURE, "Error in NIDAQ stepper output\n");
  }
  CHK( nsleep(100) );
  PING(1);
  return SUCCESS;
}
```

Figure 4.2: An excerpt from the digital output module of my experiment server stack. Most of the C code is error checking and tracing macros. The hardcoded delay time and stepper-specific error message are symptoms of my previously poor programming practices. `Write_WriteDigPort` is a simplifying wrapper around `DAQmxWriteDigitalU32` from the examples bundled with NI-DAQmx.

a message passing server with experiment control and hardware interface modules connected via sockets.

As the experiment server evolved, I started running into problems. The overhead of sending all the data through sockets to generic hardware interface modules was larger than I had naïvely expected. I also had trouble with multithreaded socket code on Cygwin, and decided to drop Microsoft Windows altogether in favor of an open source operating system.

4.1.3 Comedi

After transitioning to Linux-based systems, I could no longer use NI-DAQmx (which only supported Microsoft Windows). Luckily, the Comedi project already provided open source driver code for our DAQ card (an NI-PCI-6052E). Comedi (from "Control and Measurement Device Interface") is a general purpose library for interacting with DAQ devices, and supports a wide range of hardware. When I moved to Comedi, it was a stand-alone kernel module, but since November 2008 it has been included in the Linux source as a staging driver.

Comedi development goes back to 2000, so by the time I arrived things were already pretty stable. I submitted a small patch to support simultaneous analog input/output triggering on National

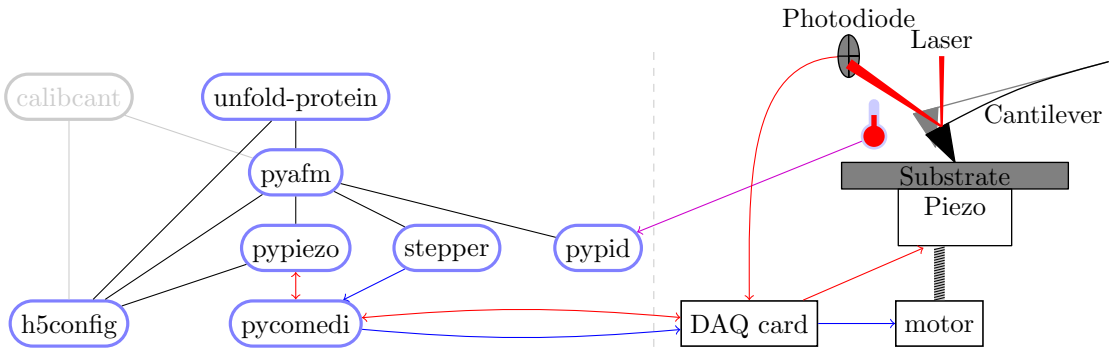


Figure 4.3: Dependency graph for my modular experiment control stack. The unfold-protein package controls the experiment, but the same stack is used by calibcant for cantilever calibration (Fig. 5.1). The dashed line ( ) separates the software components (on the left) from their associated hardware (on the right). The data flow between components is shown with arrows. For example, the stepper package calls pycomedi, which talks to the DAQ card, to write digital output that controls the stepper motor (, Section 4.3.2). The pypiezo package, on the other hand, uses two-way communication with the DAQ card ( ), writing driving voltages to position the piezo and recording photodiode voltages to monitor the cantilever deflection (Section 4.2.2). The pypid package measures the buffer temperature using a thermocouple inserted in the fluid cell (, Section 4.3.3). I represent the thermocouple with a thermometer icon ( ), because I expect it is more recognizable than a more realistic .

Instruments cards, and started building my stack.

4.2 The pyafm stack

In order to reduce future maintenance costs, I have based my stack as much as possible on existing open source software, and split my stack into reusable components where such components might appeal to a wider audience. From the bottom up, pycomedi wraps the Comedi device driver for generic input/output, pypiezo builds generic piezo-control logic on top of pycomedi, pyafm combines a pypiezo-controlled piezo with a stepper-controlled stepper motor and pypid-controlled temperature controller, and unfold-protein adds experiment logic to pyafm to carry out velocity-clamp force spectroscopy (Fig. 4.3).

4.2.1 Pycomedi

After my experience with C (Section 4.1.2), I knew I wanted a higher level language for the bulk of my experiments. Comedi already had SWIG-generated Python bindings, so I set to work creating pycomedi, an object-oriented interface around the SWIG bindings. The first generation pycomedi

```
from pycomedi.device import Device
from pycomedi.channel import DigitalChannel
from pycomedi.constant import SUBDEVICE_TYPE, IO_DIRECTION

device = Device('/dev/comedi0')
device.open()

subdevice = device.find_subdevice_by_type(SUBDEVICE_TYPE.dio)
channels = [subdevice.channel(i, factory=DigitalChannel)
            for i in (0, 1, 2, 3)]
for chan in channels:
    chan.dio_config(IO_DIRECTION.output)

def write(value):
    subdevice.dio_bitfield(bits=value, write_mask=2**4-1)
```

Figure 4.4: A four-channel digital output example in pycomedi (from the stepper doctest). Compare this with the much more verbose Fig. 4.2, which is analogous to the `subdevice.dio_bitfield()` call.

contains code to convert piezo motion (in meters) to DAC output voltages (in bits), an h5config-based framework for automatically configuring pycomedi channels and piezo axes, and surface detection logic.

Because of the tight coupling needed between piezo motion and cantilever deflection detection for synchronized ramps, the basic `Piezo` class can be configured with generic pycomedi input channels. In practice, only the cantilever deflection is monitored, but if other pypiezo users want to measure other analog inputs, the functionality is already built in.

The surface detection logic is somewhat heuristic, although it has proven quite robust in practice. Given a particular piezo axis, target deflection, number of steps, and an allowed piezo range, the procedure is:

1. Ramp the piezo from its current position away to its maximum separation $z_{\max}$.
2. Step the piezo in towards its minimum separation, checking the deflection after each step to see if the target deflection threshold has been crossed. This is the high-contact piezo position $z_{\min}$.
3. Ramp the piezo away to its maximum separation $z_{\max}$. Because of protein on the surface, the

```
class Unfolder (object):
    # ...
    def run(self):
        """Approach-bind-unfold-save[-plot] cycle.
        """
        ret = {}
        ret['timestamp'] = _email_utils.formatdate(localtime=True)
        ret['temperature'] = self.afm.get_temperature()
        ret['approach'] = self._approach()
        self._bind()
        ret['unfold'] = self._unfold()
        self._save(**ret)
        if _package_config['matplotlib']:
            self._plot(**ret)
        return ret
```

Figure 4.5: The main unfolding loop in unfold-protein. Compare this with the much more opaque pull phase in Fig. 4.1.

`AFM.stepper_approach` quickly positions the surface within piezo-range of the cantilever tip by stepping in (with the stepper motor) until the cantilever deflection crosses a target threshold. The piezo extension is kept constant during the approach, but a single stepper step only moves the surface ~ 170 nm, and our cantilevers can safely absorb deflections on that scale.

`AFM.move_just_onto_surface` is a more refined version of `AFM.stepper_approach`. This method uses pypiezo's surface detection algorithm to locate the surface kink position z_{kink}, and adjusts the stepper in single steps until the measured kink is within two stepper steps (~ 340 nm) of the centered piezo position. Then it shifts the piezo to position the cantilever tip at an exact offset from the measured kink. This precise positioning is used for running calibcant's bumps (Section 5.4.1), but the per-step piezo manipulation makes long distance approaches much slower than `AFM.stepper_approach`.

4.2.4 Unfold-protein

Capping the experimental control stack, unfold-protein adds the actual experiment logic to the lower level control software. The abstractions provided by the lower level code make for clean, easily adaptable code (Figs. 4.5 and 4.6).

```
class UnfoldScanner (object):
    # ...
    def run(self, stepper_tweaks=True):
        self._stop = False
        _signal.signal(_signal.SIGTERM, self._handle_stop_signal)
        self.unfolder.afm.move_away_from_surface()
        self.stepper_approach()
        for i in range(self.config['velocity']['num loops']):
            _LOG.info('on loop {} of {}'.format(
                    i, self.config['velocity']['num loops']))
            for velocity in self.config['velocity']['unfolding velocities']:
                if self._stop:
                    return
                self.unfolder.config['unfold']['velocity'] = velocity
                try:
                    self.unfolder.run()
                except _ExceptionTooFar:
                    if stepper_tweaks:
                        self.stepper_approach()
                    else:
                        raise
                except _ExceptionTooClose:
                    if stepper_tweaks:
                        self.afm.move_away_from_surface()
                        self.stepper_approach()
                    else:
                        raise
                else:
                    self.position_scan_step()
```

Figure 4.6: The scanning loop unfold-protein. Unfolding pulls are carried out with repeated calls to `self.unfolder.run()` (Fig. 4.5), looping over the configured range of velocities for a configured number of cycles. If `stepper_tweaks` is `True`, the scanner adjusts the stepper position to keep the surface within the piezo's range. After a successful pull, `self.position_scan_step()` shifts the piezo in the x direction, so the next pull will not hit the same surface location.

```
import h5config.config as _config

class AxisConfig (_config.Config):
    "Configure a single piezo axis"
    settings = [
        _config.FloatSetting(
            name='gain',
            help=(
                'Volts applied at piezo per volt output from the DAQ card '
                '(e.g. if your DAQ output is amplified before driving the '
                'piezo),')),
        _config.FloatSetting(
            name='sensitivity',
            help='Meters of piezo deflection per volt applied to the piezo.'),
        # ...
        _config.ConfigSetting(
            name='channel',
            help='Configure the underlying DAC channel.',
            config_class=OutputChannelConfig,
            default=None),
        # ...
        ]
```

Figure 4.7: Portions of the configuration class for a single piezo axis (from pypiezo, Section 4.2.2). The more generic analog output channel configuration is nested under the `channel` setting.

4.3 Auxiliary packages

The previous section covered the core of the experiment stack (Section 4.2), but skipped over some of the more peripheral packages.

4.3.1 h5config

The h5config package makes it easy to save and load configuration classes from disk. After populating base configuration classes with parameters (Fig. 4.7), h5config automatically generates HDF5 and YAML backends for saving and loading that class.

Basic configuration types include booleans, integers, floating point numbers, enumerated choices, and freeform text. There is also support for lists of these basic types (e.g. lists of integers). The key feature is nesting configuration classes. This means that your higher level tools can have their own configuration settings and also include the configuration settings for their lower level components.

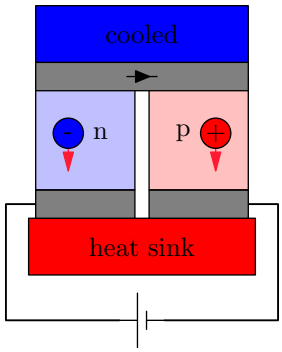


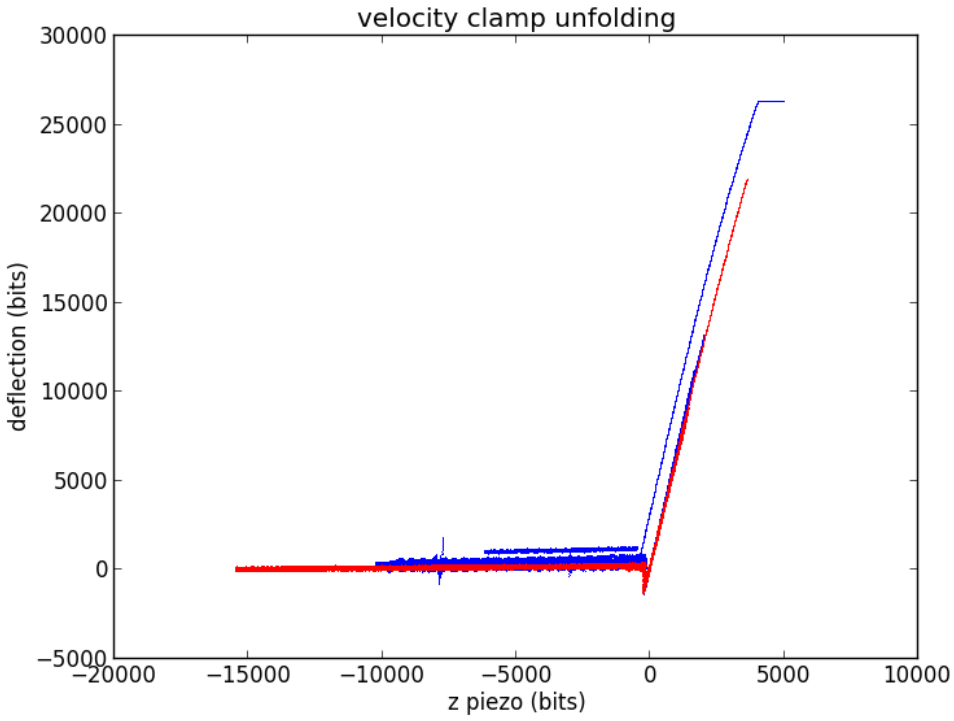
Figure 4.9: A Peltier functions by applying a voltage to regions of p- and n-type semiconductor in series. Conduction in n-type semiconductors is mainly through thermally excited electrons and in p-type semiconductors is mainly through thermally excited holes. Applying a positive voltage as shown in this figure cools the sample by constantly pumping hot conductors in both semiconductors towards heat sink, which radiates the heat into the environment. Reversing the applied voltage heats the surface.

4.4 Discussion

With the radical shift from LabVIEW and Microsoft Windows over to Comedi and Linux, it is a good idea to compare my new experiment control software with the earlier stack. Because the fundamental procedure in my experiments is the velocity-clamp pull (Section 2.4), I used both approaches in quick succession to collect pulls. Because the stacks diverge after the PCI DAQ card, I was able to collect several pulls using my setup, power down the Linux computer, swap the PCI card into the Windows computer, power up, and collect several pulls using the Windows stack on top of the exact same hardware.

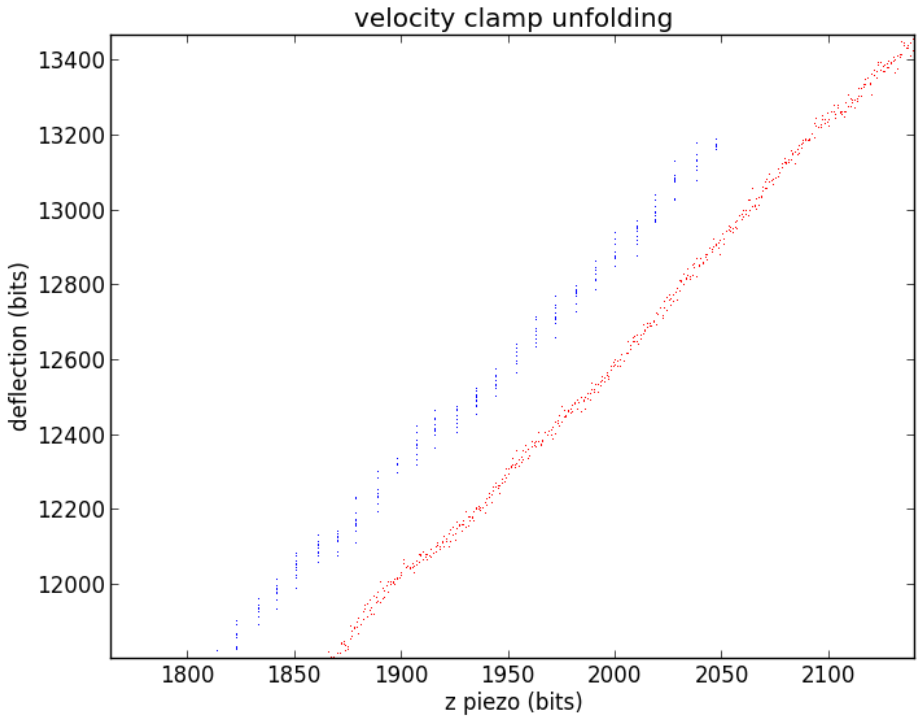
Because the goal of these experiments was to compare the two software stacks, the comparison was carried out using our standard AFM cantilevers and gold surface, but distilled water was used instead of PBS and no protein was bound to the surface. This gives a simpler system with fewer distracting features. As shown in Figs. 4.10 and 4.11, large-scale features are identical, with similar contact slopes and non-contact noise.

Although grossly similar, the two stacks do have some statistically significant differences. The slope of the contact region for the LabVIEW/Windows stack (excluding the out-of-deflection-range outlier) is 6.59 ± 0.13, while the Comedi/Linux stack slope is 6.17 ± 0.03 (both in deflection bits per



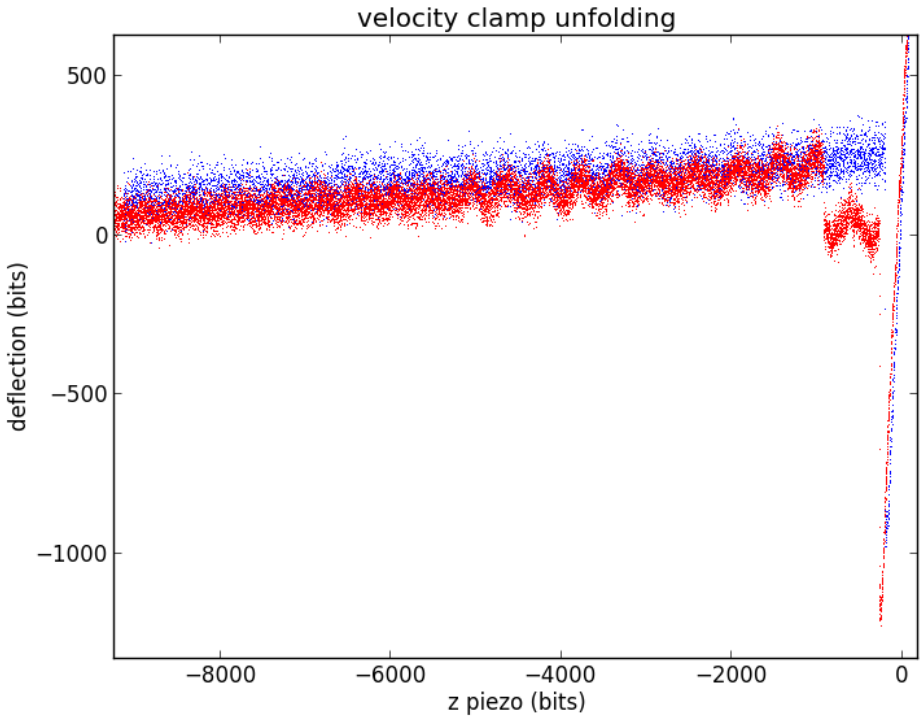
(a)

Figure 4.10: (a)Several velocity clamp pulls using both the LabVIEW/Windows stack (blue) and the Comedi/Linux stack (red). The contact voltage and pulling distance were not synchronized between the two experiments, and the raw data has been shifted to locate the contact point at the origin. One LabVIEW/Windows curve has a flat deflection in the high-contact region, where the laser was deflected beyond the photodiode's working range. This is probably due to a high approach setpoint, followed by surface drift during the binding phase, but is not relevant to the stack comparison.



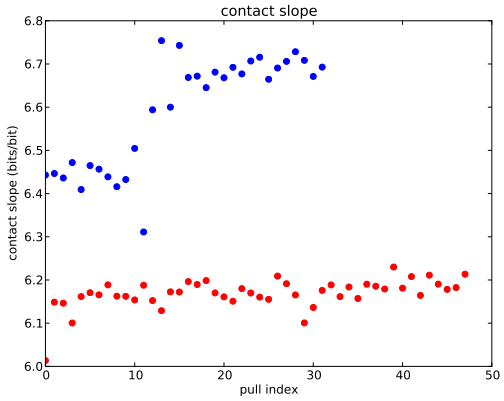
(b)

Figure 4.10: (b)The contact region from a single pull from (a). The LabVIEW/Windows stack (blue) takes 0.5 nm piezo steps with ten deflection reads at each step. The Comedi/Linux stack (red) makes a single read per step, but can take as many small steps as possible within DAQ card's memory buffer, frequency, and precision limitations. For 1 μm/s pulls, a stepping/sampling frequency of 50 kHz generated steps that were less than one DAC bit wide.

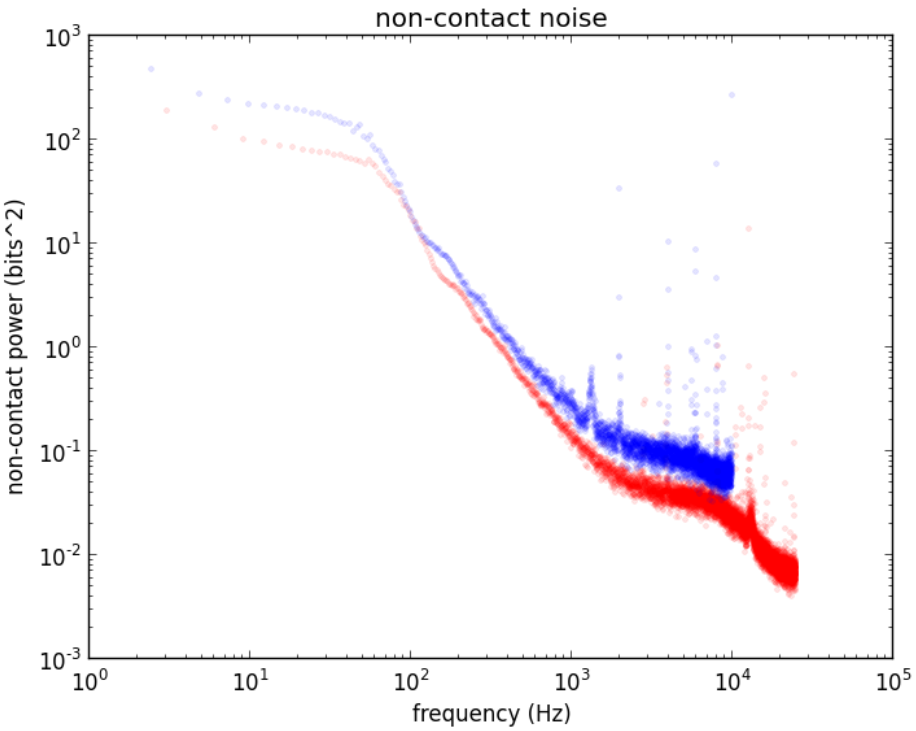


(c)

Figure 4.10: (c)The non-contact region from a single pull from (a) using both the LabVIEW/Windows stack (blue) and the Comedi/Linux stack (red). The signal oscillates because the AFM is sitting directly on the lab bench, our usual isolation mechanisms being unavailable when these curves were recorded. All protein unfolding experiments were carried out with isolation, so the vibration was not a problem in those cases.



(a)



(b)

Figure 4.11: (a)Contact slope for the pull in Fig. 4.10 using both the LabVIEW/Windows stack (blue) and the Comedi/Linux stack (red). The low-slope outlier is from the pull with out-of-range deflection. (b)Power spectral densities (PSDs) of the non-contact noise for the pulls from 4.10a. To produce this image, the PSD of the non-contact data extracted from 4.10a was averaged for each software stack. The number of points in the non-contact region truncated to the nearest power of two (for efficient fast Fourier transformation), which has the convenient side effect of aligning the frequency axis for easy cross-pull averaging.

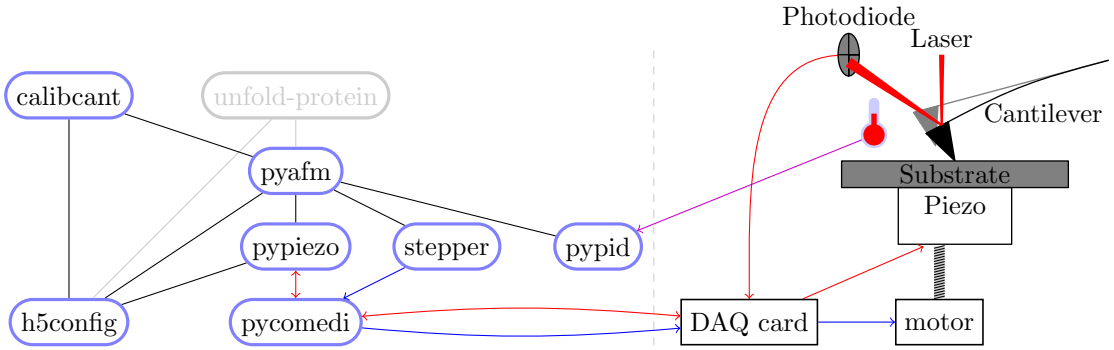


Figure 5.1: Dependency graph for calibcant, which shares the pyafm stack with unfold-protein(Fig. 4.3). The only difference is that the "brain" module controlling the stack has changed from unfold-protein to calibcant.

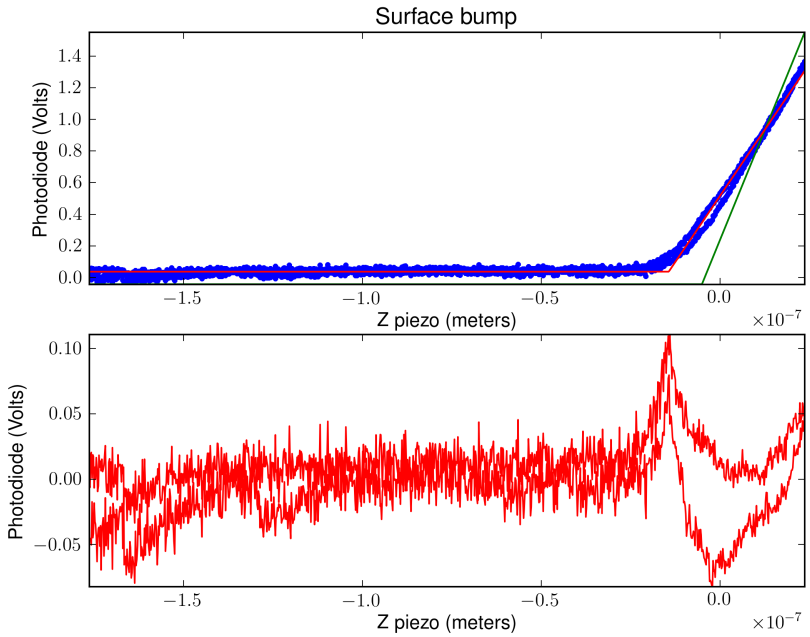
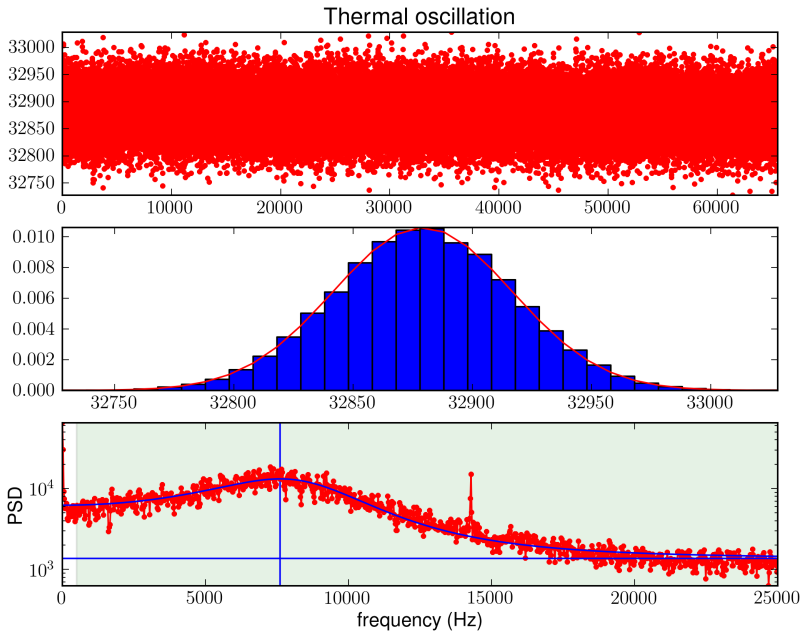
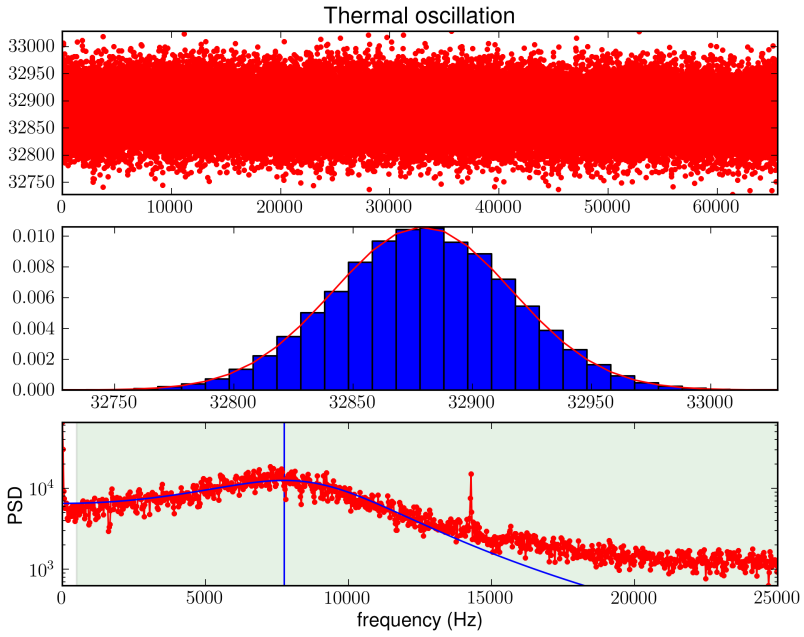


Figure 5.2: Measuring the photodiode sensitivity σ_p by bumping the cantilever tip on the substrate surface. In the first panel, the blue dots are experimental data, the green line is the heuristic guess at initial fitting parameters, and the red line is the optimized fit. The second panel shows the residual (measured data minus modeled data) for the bump. In both panels, there are two deflection measurements at each position, one taken during the approach phase and another taken during the retraction phase. This is the first bump from the 2013-02-07T08-20-46 calibration.



(a)

Figure 5.3: (a)Measuring the cantilever's thermal vibration. The top panel shows the raw time series data in bins, the middle panel shows the distribution of bin values with a Gaussian fit, and the bottom panel shows the $\mathrm{PSD}_f(V_p, f)$ with a fit following Eq. (5.76). The constant offset P_{0f}, drawn as the horizontal line in the third panel, accounts for white noise in the measurement circuit[26]. The vertical line marks the peak frequency $f_{\max}$ (Eq. (5.80)). Only data in the blue region was used when computing the best fit. This is the first vibration from the 2013-02-07T08-20-46 calibration, yielding a fitted variance $\left\langle V_p(t)^2 \right\rangle = 96.90 \pm 0.99\ \mathrm{mV}^2$. The narrow spike around 14.3 kHz is not due to the cantilever's thermal vibration, and rejecting noise like this is the reason we use a frequency-space fit to calculate the thermal deflection variance.



(b)

Figure 5.3: (b)This is the same data as in (a) fit with Eq. (5.3), yielding a fitted variance $\langle V_p(t)^2 \rangle = 120.92 \pm 0.90$ mV^2. The third panel is very similar to figure 2 in Florin et al.[27], but they do not go into further detail on the method or model. They may be fitting their data to Eq. (5.6), see Section 5.2. Another similar figure is in Hutter and Bechhoefer[28].

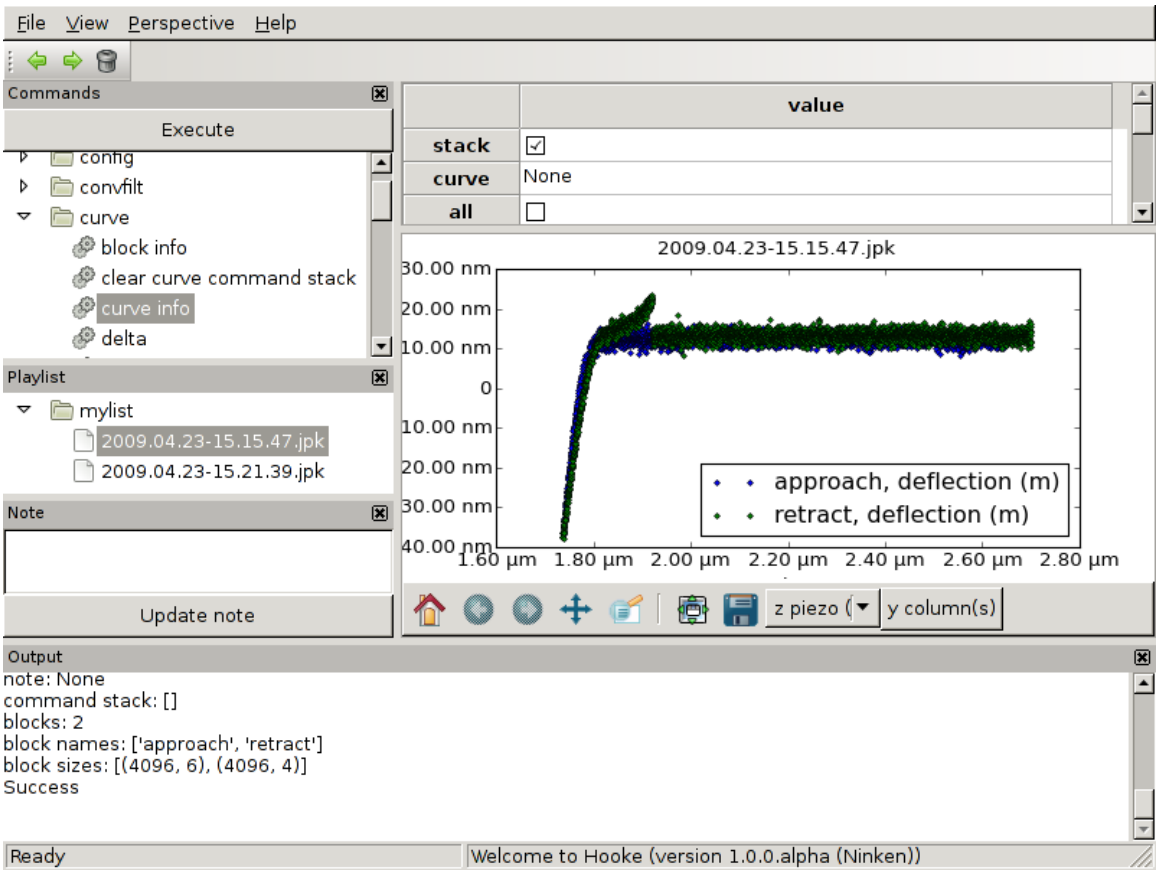


Figure 6.2: Creating a playlist with two JPK files in the graphical Hooke interface. You can see the output of the last *curve info* call, which matches the output from the command line version (Fig. 6.1). This screenshot is a bit cramped (to fit on a printed page), but the wxWidgets GUI toolkit provides automatic support for interactively rearranging and resizing panels. The tree of commands is in the upper left corner. After you select a command, the table of argument in the upper right corner is populated with default values, which you can adjust as you see fit. The *Playlist* panel provides an easier interface for navigating to different playlists and curves than the using *jump to playlist* and *jump to curve* commands.

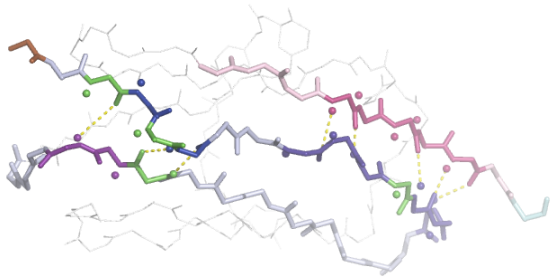


Figure 7.1: The backbone of I27 showing the eight key hydrogen bonds responsible for the critical unfolding force. Glutamic acids are highlighted in green. Based on Lu and Schulten[29] Fig. 1b. For a ribbon diagram of I27 showing the β-sheets, see Fig. 2.3. This figure was also generated with PyMol.

is what we expect due to destabilized hydrogen bonding.

Modeling I27 as a Bell-model unfolder, we can use sawsim to find the Bell parameters that best fit these experimental unfolding histograms (Sections 3.2.2 and 3.3.4). The results in Fig. 7.3 show that the best fit for standard PBS was with $\Delta x_u = 0.132$ nm and $k_{u0} = 0.222$ s^{-1}. In the Ca^{2+} buffer, the best fit was with $\Delta x_u = 0.123$ nm and $k_{u0} = 0.450$ s^{-1}.

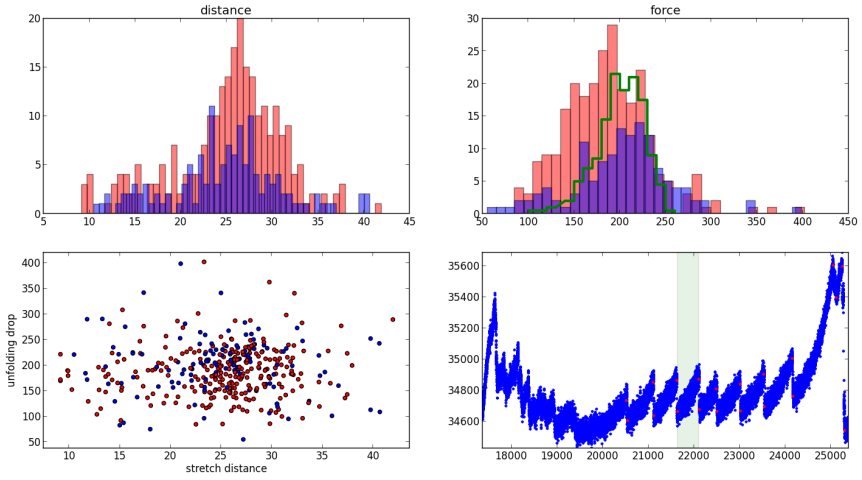


Figure 7.2: I27 runs from 2013-03-04 with (red) and without (blue) an extra 0.5 M Ca^{2+}. Clockwise from the upper left, we have the distance (in nm) between peaks, the unfolding force (in pN), and example force curve, and a scatter plot of unfolding force (in pN) versus the distance between peaks. All of the pulls were taken with the same Olympus TR400-PSA cantilever with a pulling speed of 1 μm/s. The green histogram drawn over the unfolding force histograms is I27 unfolding data in PBS with 5 mM DTT from Carrion-Vazquez et al. [6], rescaled by a factor of $\frac{1}{2}$ because they had more unfolding events.

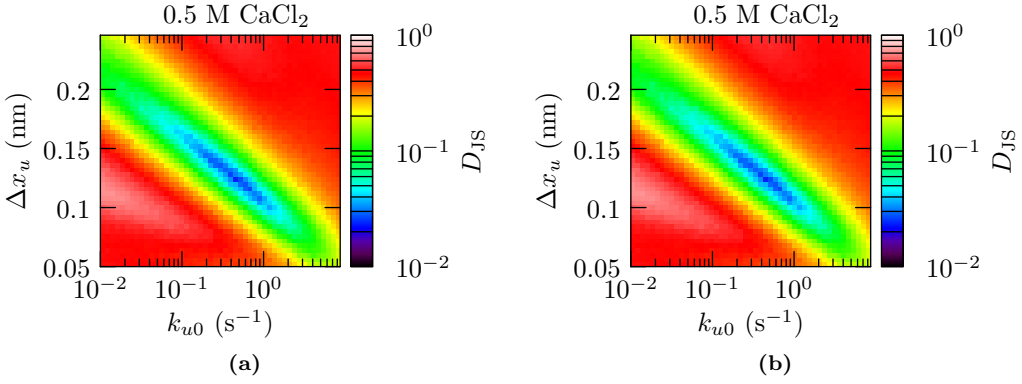


Figure 7.3: (a) Model fit quality for the standard PBS unfolding histogram data shown in Fig. 7.2. (b) Model fit quality for the Ca^{2+}-enhanced PBS unfolding histogram data. The best fit parameters occur when the Jensen–Shannon divergence is minimized (at the bottom of these valleys, Section 3.3.4).

Appendix A: Cantilever calibration

A.1 Contour integration

As a brief review, some definite integrals from $-\infty$ to ∞ can be evaluated by integrating along the contour $\mathcal{C}$ shown in Fig. A.1.

A sufficient condition on the function $f(z)$ to be integrated, is that $\lim_{|z|\to\infty}|f(z)|$ falls off at least as fast as $\frac{1}{z^2}$. When this is the case, the integral around the outer semicircle of $\mathcal{C}$ is 0, so the $\int_{\mathcal{C}} f(z)\mathrm{d}z = \int_{-\infty}^{\infty} f(z)\mathrm{d}z$.

We can evaluate the integral using the residue theorem,

$$\int_{\mathcal{C}} f(x)\mathrm{d}z = \sum_{z_p\in\{\text{poles in } \mathcal{C}\}} 2\pi i \,\mathrm{Res}\left(z = z_p, f(z)\right) \ , \tag{A.1}$$

where for simple poles (single roots)

$$\mathrm{Res}\left(z = z_p, f(z)\right) = \lim_{z\to z_p} (z - z_p) f(z) \ , \tag{A.2}$$

and in general for a pole of order n

$$\mathrm{Res}\left(z = z_p, f(z)\right) = \frac{1}{(n-1)!} \cdot \lim_{z\to z_p} \frac{\mathrm{d}^{n-1}}{\mathrm{d}z^{n-1}} \left[(z - z_p)^n \cdot f(z)\right] \ . \tag{A.3}$$

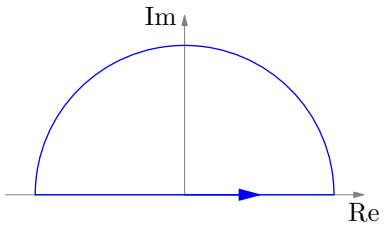


Figure A.1: Integral contour $\mathcal{C}$ enclosing the upper half of the complex plane. If the integrand $f(z)$ goes to zero "quickly enough" as the radius of $\mathcal{C}$ approaches infinity, then the only contribution comes from integration along the real axis (see text for details).