Graphical User Interface for Intermodulation
Atomic Force Microscopy

Borg, Malin  Ghattas, Elias  Syla, Ardian
malinbor@kth.se  nizareg@kth.se  ardian@kth.se

SA104X Degree project in engineering physics, first level
Department of Applied Physics, Royal Institute of Technology
Supervisors: Forchheimer, Daniel,
Professor Haviland, David and Platz, Daniel

May 26, 2011
Abstract

In atomic force microscopy (AFM) a cantilever, with a sharp tip attached to its free end, is scanned over a surface. Forces from the surface affect the tip causing cantilever deflection, which is registered by a detector. This information is then used to create a topographical image of the surface on nanoscale. We were a part of a project that developed a new type of AFM, namely intermodulation atomic force microscopy (IMAFM). It is based on dynamic AFM, but instead of one drive frequency, two or more frequencies are used. This generates more information from the sample. Our part in this project was to improve the software already in use. This will facilitate future simulations and experiments; and also utilize the available information in a new way.
Sammanfattning

1 General Introduction

1.1 The concepts of AFM; Static and Dynamic

The general concept behind an atomic force microscope (AFM) can be described as following; a cantilever, with a very sharp tip attached to its free end, scans the surface of a sample. Forces that act between the sample surface and the tip causes the cantilever to deflect. As the tip is scanning over the sample - or the sample is scanned under the tip - a detector measures the cantilever deflection, as illustrated in figure 1. Using the deflection data, a computer generates a topographic map of the surface.

Today, there exists a multitude of AFM imaging methods, and they can be separated into two groups: static and dynamic. In static mode, the tip makes gentle physical contact with the sample. When the tip then traces the surface, the cantilever will accommodate to topography changes due to the contact force. The disadvantages of this method is that it damages the tip as well as the sample. Albeit these damages might be relatively small they add up and will force a change of the cantilever tip more often than in dynamic AFM. On the other hand, the advantages of static AFM is that even if a sample is covered in a thin layer of liquid, it will still represent the surface correctly.

In dynamic AFM, the cantilever is held a small distance above the sample while oscillating at its resonance frequency. Thus it does not have continuous contact with the surface. Dynamic AFM has therefore the advantage that it does not suffer from the same amount of tip or sample degradation as contact mode. But the disadvantages are that it will have trouble resolving surfaces that have small regions covered in liquid. The images of the liquid and the sample will mix and thereby cause a faulty image [2].

1.2 Intermodulation AFM

In this project we have been working with a new type of AFM called intermodulation atomic force microscopy (IMAFM). This method is similar to standard dynamic AFM (e.g. tapping mode AFM) with one major difference. In regular tapping mode AFM, a single drive frequency is used to
oscillate the cantilever \[2\]. ImAFM applies two or more drive frequencies to the cantilever, causing its oscillations to become non-linear. The two drive frequencies mix with each other and generate new frequencies. These make up a phenomenon, in nonlinear systems driven with more than one frequency, known as Intermodulation products (IMPs). These IMPs are usually unwanted in most engineering contexts, as they are considered an effect that can only be used to measure the signal distortion due to non-linearity. These IMPs are used to create a very sensitive, high information bandwidth mode of AFM; this is what we call IMAFM \[3\].

The experimental setup used in IMAFM can be seen in figure 2. This setup consists of three parts: the AFM with corresponding controller, the Nanoscope, and a custom made electrical component called IMP2-32. This component is what sends the two drive frequencies to the AFM microscope. In return, it receives information about the cantilever deflection from the photodiode in the AFM. When the cantilever has reached the end of a scanned line or the end of the sample, the Nanoscope sends a signal to the IMP2-32. Feedback is then sent back to the Nanoscope, which uses this information for adjusting the cantilever position.

From the lockin intermodulation module we get a discrete set of frequency data points with given amplitudes and phases. When these are Fourier transformed they generate a spectrum of frequencies in the form of bins made up of integers of the two drive frequencies. If there were to be created points in the transform of frequencies that are not integers of these drive frequencies then some energy would “leak” into the adjacent bins causing a distortion in the transform called “Fourier leakage” as seen in the figures 3 and 4.
Figure 3: Picture illustrating leakage free frequency space.

Figure 4: Picture illustrating Fourier leakage.
1.3 Our Project; How and Why

IMAFM is a new method with great potential when it comes to surface scanning. With further optimization it could, in a single scan, acquire images with even higher sensitivity and greater information content. Thus IMAFM would improve scan times, but this would be of no use if the extra attachments, required to improve a regular atomic force microscope to IMAFM, became a new time sink. The aforementioned new type of electrical component, the IMP2-32, is not sufficient to complete the experimental setup. This is why a software called ImAFM suite, that would handle the information sent by the IMP 2-32 and make it easy to use, was developed. In order to make this setup even better and facilitate the experiments, we have been developing some applications for the ImAFM suite.

In our project we intend to improve two aspects important for experimentalists; the control of the setup and the analysis of the data acquired from the ImAFM. To improve the control, two modules were constructed to control data and to coordinate between all the components. A virtual oscilloscope was developed to get a real time perspective of the distortions in the cantilever movements. The communication between the Nanoscope and the IMAFM suite was improved by creating a program that could remote control the Nanoscope. In order to analyze the height and force data received from the ImAFM, a third and final module was made to create images of the scanned surface and to visualize the interaction forces between the tip of the cantilever and the surface.

2 Oscilloscope Integration

An oscilloscope is a device used to measure and display voltage waveforms and is pivotal in signal processing, due to its versatility and quick response time. In our setup, an oscilloscope is attached to the microscope in order to monitor the output signals. This yields the ability to verify the success of the scanning. A virtual oscilloscope could be integrated to the software, thus eliminating the need for a physical oscilloscope.

In order to build a virtual oscilloscope, one must first understand the different theories concerning signal processing in both frequency and time space. The major difference between these two spaces is that the frequency space is a discrete one, with specific frequencies being given at discrete points in time; whereas the time space is constructed to be pseudo continuous of discretely sampled signals. To move between these two spaces, the frequency data is transformed to time data with the inverse Fourier transforms.

Because the oscilloscope operates in real time, the data needs to be continuously processed. For that purpose, some time saving tricks were applied, as described in [4]. When the two drive frequencies are inserted in the program, they run though a series of steps aimed to reduce the processing
time and maximizing the data, so to improve the quality of the output signal.

First, a number of zeros is appended in the space preceding the frequencies. This is called “zero padding” and is aimed to increase the number of samples, which smooths the plotted functions. Second, the frequency data needs to be transformed to time data by the inverse Fourier transform. Inverse Fast Fourier Transform (IFFT) algorithms are used to keep the transformation time to a minimum. The IFFT works by breaking down the discrete signal into a sequence of smaller signals and then transforming them individually. This decreases the transformation time as the Fourier transform is dependent on the length of the data set to be transformed.

Third, the amount of data gathered by the oscilloscope was staggering, leading to a time-consuming process of processing, transforming and plotting the data. To make sure that the output data is displayed in real time, an algorithm to reduce the Fourier-transformed data had to be constructed. The basic idea behind the data reducing algorithm was to divide the input data into a number of intervals, finding the minimum and maximum points in each interval and finally plotting these chosen values, as seen in figure 5. This reduced the processing time drastically.

![Figure 5: Illustration of the reduction algorithm.](image)

Finally, so-called “threads” [5] were used. These threads are functions used for multi-processing of data, i.e. that a program could run two independent processes at the same time. In the oscilloscope program, a thread was used to simultaneously process data and plot it (it would actually process one data point, plot it, then jump to the next data point). The final oscilloscope was put into a GUI. There, the user can choose the number of intervals, the size of the zero padding and the preferred data space, as shown
in figures 6 and 7. The first figure displaying time space and the second, frequency space.

Figure 6: Output signal, in time space, in a working virtual oscilloscope.

Figure 7: Output signal, in frequency space, in a working virtual oscilloscope.
3 Visualization of 3D Force Field

3.1 Introduction

Following a successful scanning of a substrate, the AFM will produce a 2D topographical surface image. This along with a data set containing the force on the cantilever from the surface, at the point \((x,y,z)\). In order to make sense of these obtained force values, a 3D volume of the force data is made over the surface image. This creating a formidable tool for visualizing the cantilever-surface interaction forces, which in turn provide vital information of the properties of the material that is being scanned. The use of force field maps is a new, but already existing in the world of AFM [6].

The process of developing our force field map consists of two main steps; processing the collected data and making a 3D plot out of it. To ensure that the visualization is accurate, it is imperative that the topographical information is as correct as possible; this meaning that there should be no tilt in the substrate whilst it is being scanned, this is nearly impossible to achieve with an AFM. One could on the other hand find the tilt in the produced surface image and then adjust the image accordingly. The adjusted topographical data, along with the corresponding force is then plotted to get the 3D force map. This specific plotting technique is a unique feature of IMAFM, since we are not only plotting the force in 3D, but also positioning it correctly over the various surface features.

3.2 Planarization

In order to planarize the data generated in the IMAFM, a general equation of the topographic image is needed. To define this equation we need to select an area on the image, where the surface is the flattest. The equation is then produced using basic linear algebra. All of this is done using a GUI, where the surface image, along with the height data is displayed. As the topography is plotted, one can easily identify and select a flat area. From this selected area, and the accompanied height data, all the necessary data is obtained. These data sets are then processed to produce the general plane equation, using the algorithms described below. This plane is then subtracted from the entire surface, reducing the tilt of height data. The new, modified topography is then ready to be used in the 3D plotting.

A plane in three dimensions is defined with the help of at least three arbitrary points in space. These points can create a plane by taking the distance between two of them and the normal to the plane. The general form of a plane in two dimensions is \(z = c_0 + c_1 \cdot x + c_2 \cdot y, \forall x, y \in \mathbb{R}\). The objective is now to calculate the constants in this equation. The most straightforward method to achieving this is by solving the set of equations in a least-squares sense. Least squares is a linear regression of a set of data, solved by minimizing the error term \(e\) in the regression of the form
\[ z - A \cdot c = e \]. This is done by simply solving the normal equation, \[ z - A \cdot c = 0 \], in the following manner:

\begin{align*}
z - A \cdot c &= 0 & \text{(1)} \\
A \cdot c &= z & \text{(2)} \\
c &= (A^T \cdot A)^{-1} \cdot A^T \cdot z & \text{(3)}
\end{align*}

To invert a matrix of arbitrary form in a quicker way, one could resort to a more general definition of matrix inversion. The Moore-Penrose pseudo inverse algorithm is the one used to solve the least squares problem in question, by means of singular value decomposition. Luckily there is a built in function in the Python library Numpy that utilizes the algorithm, namely \texttt{numpy.linalg.pinv()}. The reason that we choose our data from the part on the image that is the flattest, is that it contains few topographical changes. Sharp changes in the data used in the least squares will result in bigger tilt than desired. Now that the general plane equation has taken form, we could adjust the original data by reducing the plane values from it. The figure 8 below show a schematic view over the planarization process.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{planarization_process.png}
\caption{A schematic view over the planarization process.}
\end{figure}

\subsection{Creating the Visualization}

When faced with the question of how to turn the force and surface data into a 3D plot that is easily embedded into an existing Python application, Mayavi (version 3.4.1) \cite{mayavi} provides a straightforward solution. Mayavi is an interactive Python tool that creates elaborate 3D plots and visualizations from scientific data. These properties makes it a pivotal tool in the process of turning our force and surface data into a realistic plot. The finished visualization is then embedded into a wxPython GUI and connected with the planarization application, creating a interactive tool for both the data preparation and the resulting 3D plot.
The first step in creating the program was to make a class deriving from the Traits class [8]. Letting the class inherit HasTraits, opens up for the possibility of the user interactively changing the data and the properties of the visualization itself. In order to make the created data objects take form, the Mayavi module enthought.mayavi.mlab [9], called mlab, was used. The mlab module provides an object oriented approach to 3D visualization and gives the user access to the powerful features of Mayavi. An attribute called MlabSceneModel will then act as an container for the mlab model and its objects. This structure can be seen in figure 9.

Visualizing the surface and the force field was done by turning the inputted data into mlab objects. The surf() [10] function was applied to the height values; this creating a three-dimensional surface out of the two-dimensional elevation matrix. Plotting the force field required a two step process. First, turning the force data into a scalar field by using scalar_field() [11] and then transforming the obtained field into a volume with volume() [12].

```
scalar_field=self.scene.mlab.pipeline.scalar_field(force)
volume_plot=self.scene.mlab.pipeline.volume(scalar_field)
surface_plot = self.scene.mlab.surf(surface)
```

In the code above, one can see that the functions scalar_field() and volume() comes from the entity called pipeline. This because Mayavi build visualizations by assembling these pipelines. The pipelines are the part of Mayavi that loads the data, by using data sources [14]. From there, the data can also be transformed by filters and visualized by modules. The pipelines are also where one can explore the full power of Mayavi. Applying a mlab function builds complex pipelines while choosing the appropriate sources, filters and modules, but the mlab function only explores a few of the available possibilities. Typing the mlab.show.pipeline command displays the current pipeline.

The pipelines where used for most of the work after creating the field and surface plot. Firstly, the volume was sliced by applying image_plane_widget() [17] to the scalar field that constitutes the volume. A plane would then appear at a set value, aligned with a specific axis, displaying smaller details in a preferred region. Users can then tilt and move the plane around,
all after specific preferences. The scalar field derived from the force field data was also used to create a scalar bar. This showing the color - force dependence and thereby emphasizing the color variations of the plot, which reflects the force value in the different points. By this, the 3D model was finished and the next step of the process took on.

wxPython [15] was used to embed the already created visualization into the Python application that would construct the corresponding GUI. A wxPanel was produced, onto which a HasTraits object was created out of the aforementioned class. And by this creating a Mayavi scene onto the wxPanel. Another wxPanel was created as a container for the check boxes and sliders that would control the planes used for the slicing. With these, one can turn the slicing on and off and, if on, select the position of the planes. Adding these controls, created problems with the plane tilting; which therefore was turned off. This was done by displaying the pipeline and setting the margin size to zero [16].

When the visualization GUI was woven together with the GUI for the planarization, it became obvious that the force field was not fully coordinated with the surface. Rotating the surface around specific axises; followed by displaying the pipeline of the volume and changing the origin of the corresponding scalar field, the surface and the force field came together. This displayed the finished 3D visualization and the corresponding GUI, seen in figure 10.

Figure 10: The finished visualization GUI with area selecting and corresponding surface - force plot.
4 Remote Control of Nanoscope Software

Scientific research today depends heavily on computers. To fully utilize the power of computers while performing experiments, software is created in conjunction with new scientific methods. Many of these methods have licensed softwares which prevents users from modifying and adapting the software. This aforementioned inflexibility causes problems when new ways of using old methods are discovered. The original software was not made for these new ideas and might not suffice and it can also make implementation of these ideas cumbersome. A solution to these problems is to remote-control the old third-party software.

The third-party propriety software, called Nanoscope, is used in parallel with the ImAFM suite during experiments. While this way of experimenting certainly is possible, it is cumbersome and not very user friendly. The purpose of this part of the project was to remotely control the Nanoscope software using Autoit, a scripting language based on BASIC. Our main idea was to have the Autoit script communicate with a python “Application Programming Interface” (API). The ImAFM suite could then call functions from the PythonAPI.

The Nanoscope software works exclusively in Microsoft Windows and uses many inherent Windows objects as a result. One of the primary objects, that is used to control the Nanoscope software, is called SysListView32. It was of utmost importance that this object could be manipulated freely. All of the manipulation we deemed necessary could be done by controlling three controllers in SysListView32. The first of these was a controller called editbox, see figure 11, which is a standard object where you highlight the controller and type the value you want to assign it. The scripts task will be to highlight the editbox and read the current value as well as having the option to send a value to the controller.

![Editbox controller](image)

Figure 11: Editbox controller.

The second object we wished to manipulate is called a combobox. It’s a drop down list with different alternatives, see figure 12. When you highlight the combobox the list appears and you are allowed to choose one of the options in the list. We want the script to copy this action as well as being able to read the current value. The third and last object is called a togglebox, see figure 13. It’s a control that has two different values and toggles between them when pressed. We want the script to be able to press these toggleboxes and swap their values.

![Combobox](image)

![Togglebox](image)

Figure 11: Editbox controller.
Figure 12: Combobox controller.
Figure 13: Togglebox controller.
AutoIt is a high level language, but lacks support for Syslistview32, which made it a cumbersome task to manipulate all these controls. The major problem that arose trying to manipulate the Nanoscope was related to the structure of the Syslistview32. As we can see in figure 14, the Syslistview32 object is organized like a table or a matrix with a clear row and column structure. AutoIt had some user-defined functions (UDF) that could manipulate the Syslistview32 to a certain extent, but lacked any kind of function that allowed highlighting of any area that was not in the first column. We were then forced to create a function that would do this, using two UDFs.

We used one function that would retrieve the position of the first column and a second function that would retrieve the width of a column. Using these two functions, we could get the first columns position and by adding its width, we would get the position of column number two. Using this as a start we then made the function general so it would not matter how many columns a Syslistview32 had, the code for this function can be found in the Appendix B.

With this self-made function, we created a script that could manipulate the controls using a mixture of built-in functions and UDFs already in place.
For the EditBox we used our highlighting function and an AutoIt built-in function that could send keyboard commands to change the value of the controller. To retrieve information from the controller we highlighted the controller and saved the current value to the clipboard; the function then returned the earlier saved value.

The combobox controller had readily made UDFs. These could be used after the combobox was highlighted, to retrieve the items of the combobox as well as selecting them. The togglebox could be manipulated using the highlighting function that we made earlier. Since it sends a mouse-click, the togglebox would toggle value when highlighted. This script was then compiled into an .exe file that could be called with different parameters for different functions. Using a regular python/numpy console, we could import the API and manipulate our third party software directly from the console. The script performed very well and in the end we had a method capable of remote controlling our Nanoscope software.

The Nanoscope software can now reliably be remote controlled from the ImAFM suite and we have made performing experiments even easier. This method reduces human errors while performing scans with IMAFM, this is due to some physical limitations of the IMAFM compared to regular AFM. Our Nanoscope software, that was made for regular AFM, can change a couple of properties of the microscope and it also has built in limitations so the user can receive adequate results while using the AFM. Due to the differences of AFM and IMAFM, the physical limitations differ and the Nanoscope software made for AFM is not suited for IMAFM.

For instance, the control that changes the scan speed of the AFM is a problem. In regular AFM the scan speed can be chosen freely, but this is not the case when it comes to IMAFM. The two interacting drive frequencies create a beating wave form, as can be seen in figure 6. These two drive frequencies need to be contained in the discrete frequency spectrum of the discrete fourier transform, this condition places some limitations on our scan speed. When scanning we want the speed to be of such a magnitude that it corresponds to a period of our beating waveform. If the scan speed is not adjusted to fulfill this condition we will have the spectral leakage that was explained in the introduction section.

One use of our remote control program could be to automate the process of choosing appropriate scan speeds and directly send the correct scan speed after calculation and thereby reducing the possible errors that might occur due to human interaction. This could be applied to future restrictions of the same type, which would further improve the potential of the ImAFM suite as a tool customized for IMAFM. One problem regarding the Autoit script is that it is specifically made for a certain version of the Nanoscope software. This could cause some problems in areas where they use older or newer versions of the software, not to mention that this is not the only software on the market that is used to control AFMs.
5 General Discussion and Conclusions

The IMAFM is till in the experimental stage and the work today is done with the intent of improving the technique to fully utilize the amount of information gathered by IMAFM. Our project has improved the experimental environment and the possibility of analyzing the data received from the microscope. With some further testing and improvement of our virtual oscilloscope, a physical oscilloscope will not be needed in the future. This will facilitate the assembling of the experimental setup both for first-time users and experienced users as well.

A 3D visualization of the surface-tip forces provides more substantial information and a more profound understanding of the underlying substrate. Allowing remote control of the Nanoscope software reduces the possible amounts of human errors that can occur while using a software made for regular AFM (this due to the different physical limitations of AFM and IMAFM, the softwares need different limits on adjustable parameters). All of this combined, will improve the experimental setup and facilitate future usage of IMAFM.

6 Acknowledgments

We would like to thank David Haviland, Daniel Forchheimer and Daniel Platz who supervised, supported and made it all come together.

References


from enthought.traits.api import HasTraits, Instance
from enthought.traits.ui.api import View, Item
import numpy as np
from enthought.mayavi import mlab
from enthought.mayavi.mlab import *
from enthought.tvtk.pyface.scene_model import SceneModel
from enthought.tvtk.pyface.scene_editor import SceneEditor
from enthought.mayavi.tools.mlab_scene_model import MlabSceneModel
from enthought.mayavi.core.ui.mayavi_scene import MayaviScene

force_old = 1e9*np.load('ff01005.npy')
force2 = 2 * force_old[:, ::2, :128]

class MayaviView(HasTraits):
    scene = Instance(MlabSceneModel, ())
    view = View(Item('scene', resizable=True, show_label=False, editor=SceneEditor()))

    def __init__(self):
        HasTraits.__init__(self)

    def f(x,y):
        return np.sin(x*y)

    self.scene.scene.background = (1,1,1)
    self.scene.scene.foreground = (0,0,0)

    self.scalar_field=self.scene.mlab.pipeline.scalar_field(force)
    self.scalar_field.origin=[[-64., -64., 0]]
    self.volume_plot=self.scene.mlab.pipeline.volume(self.scalar_field)

    def set_surface(self, surface):
        self.surface_plot = self.scene.mlab.surf(surface)

    def set_x_slice(self, want_x_slice):
        if want_x_slice:
            self.x_slice=mlab.pipeline.image_plane_widget(self.scalar_field,
                                                        plane_orientation='x_axes', slice_index=150,)
            self.x_slice.ipw.margin_size_x=0
            self.x_slice.ipw.margin_size_y=0

    def set_y_slice(self, want_y_slice):
        if want_y_slice:
            self.y_slice=mlab.pipeline.image_plane_widget(self.scalar_field,
                                                        plane_orientation='y_axes', slice_index=150,)
            self.y_slice.ipw.margin_size_x=0
            self.y_slice.ipw.margin_size_y=0

    def set_z_slice(self, want_z_slice):
        if want_z_slice:
            self.z_slice=mlab.pipeline.image_plane_widget(self.scalar_field,
                                                        plane_orientation='z_axes', slice_index=150,)
            self.z_slice.ipw.margin_size_x=0
            self.z_slice.ipw.margin_size_y=0

    def set_x_slice(self, want_x_slice):
        if want_x_slice:
            self.x_slice=mlab.pipeline.image_plane_widget(self.scalar_field,
                                                        plane_orientation='x_axes', slice_index=150,)
            self.x_slice.ipw.margin_size_x=0
            self.x_slice.ipw.margin_size_y=0

    def set_y_slice(self, want_y_slice):
        if want_y_slice:
            self.y_slice=mlab.pipeline.image_plane_widget(self.scalar_field,
                                                        plane_orientation='y_axes', slice_index=150,)
            self.y_slice.ipw.margin_size_x=0
            self.y_slice.ipw.margin_size_y=0

    def set_z_slice(self, want_z_slice):
        if want_z_slice:
            self.z_slice=mlab.pipeline.image_plane_widget(self.scalar_field,
                                                        plane_orientation='z_axes', slice_index=150,)
            self.z_slice.ipw.margin_size_x=0
            self.z_slice.ipw.margin_size_y=0

A Appendix A - The Visualization Programs
self.volume_plot.volume_property.scalar_opacity_unit_distance = 5
else:
  self.x_slice.remove()
self.volume_plot.volume_property.scalar_opacity_unit_distance = 1

def set_y_slice(self, want_y_slice):
  if want_y_slice:
    self.y_slice=mlab.pipeline.image_plane_widget(self.scalar_field,
        plane_orientation='y_axes', slice_index=128,)
    self.y_slice.ipw.margin_size_x=0
    self.y_slice.ipw.margin_size_y=0
    self.volume_plot.volume_property.scalar_opacity_unit_distance = 5
  else:
    self.y_slice.remove()
    self.volume_plot.volume_property.scalar_opacity_unit_distance = 1

def set_x_slice_index(self, x_pos):
  self.x_slice.ipw.slice_index=x_pos

def set_y_slice_index(self, y_pos):
  self.y_slice.ipw.slice_index=y_pos

import wx
from ForcePlot import MayaviView
from enthought.mayavi import mlab

class MainWindow(wx.Panel):
  def __init__(self,parent):
    wx.Panel.__init__(self, parent, -1)
    self.mayavi_view=MayaviView()
    self.control=self.mayavi_view.edit_traits(parent=self, kind='subpanel').control
    mlab.show_pipeline()
    self.mayavi_view.volume_plot.lut_manager.scalar_bar.title = '[nN]'
    self.mayavi_view.volume_plot.lut_manager.show_scalar_bar = True
    self.Show(True)
    panel_ctrl=wx.Panel(self,-1)
    self.cb_x_slice=wx.CheckBox(panel_ctrl, -1, 'Slice with respect to x')
    self.Bind(wx.EVT_CHECKBOX, self.on_cb_x_slice, self.cb_x_slice)
    self.cb_y_slice=wx.CheckBox(panel_ctrl, -1, 'Slice with respect to y')
    self.Bind(wx.EVT_CHECKBOX, self.on_cb_y_slice, self.cb_y_slice)
self.Bind(wx.EVT_CHECKBOX, self.on_cb_y_slice, self.cb_y_slice)

self.x_slider=wx.Slider(panel_ctrl, -1, 10, 1, 250, name='x-slice:',
style=wx.SL_HORIZONTAL| wx.SL_LABELS)
self.x_slider.Disable()
self.Bind(wx.EVT_SLIDER, self.on_x_slider, self.x_slider)

self.y_slider=wx.Slider(panel_ctrl, -1, 10, 1, 250, name='y-slice:',
style=wx.SL_HORIZONTAL| wx.SL_LABELS)
self.y_slider.Disable()
self.Bind(wx.EVT_SLIDER, self.on_y_slider, self.y_slider)

szr_ctrl=wx.BoxSizer(wx.VERTICAL)
szr_ctrl.AddStretchSpacer()
szr_ctrl.Add(self.cb_x_slice, 0, wx.ALIGN_CENTER)
szr_ctrl.Add(self.x_slider, 0, wx.ALIGN_CENTER)
szr_ctrl.Add(self.cb_y_slice, 0, wx.ALIGN_CENTER)
szr_ctrl.Add(self.y_slider, 0, wx.ALIGN_CENTER)
szr_ctrl.AddStretchSpacer()
panel_ctrl.SetMinSize((150,500))
panel_ctrl.SetSizer(szr_ctrl)
szr_ctrl.Fit(self)
szr_ctrl.SetSizeHints(self)

self.szr_main=wx.BoxSizer(wx.HORIZONTAL)
self.szr_main.Add(panel_ctrl, 0, wx.EXPAND)
self.control.SetMinSize((700,500))
self.szr_main.Add(self.control, 0 , wx.EXPAND)
self.SetSizer(self.szr_main)
szr_main.Fit(self)
self.Layout()

def make_x_slice(self):
    self.mayavi_view.set_x_slice(True)

def make_y_slice(self):
    self.mayavi_view.set_y_slice(True)

def update_x_slice_index(self):
    self.mayavi_view.set_x_slice_index(self.x_slider.GetValue())

def update_y_slice_index(self):
    self.mayavi_view.set_y_slice_index(self.y_slider.GetValue())
def remove_x_slice(self):
    self.mayavi_view.set_x_slice(False)

def remove_y_slice(self):
    self.mayavi_view.set_y_slice(False)

def on_cb_x_slice(self,event):
    if self.cb_x_slice.GetValue():
        self.x_slider.Enable()
        self.make_x_slice()
    else:
        self.remove_x_slice()
        self.x_slider.Disable()

def on_cb_y_slice(self,event):
    if self.cb_y_slice.GetValue():
        self.y_slider.Enable()
        self.make_y_slice()
    else:
        self.remove_y_slice()
        self.y_slider.Disable()

def on_x_slider(self,event):
    self.update_x_slice_index()

def on_y_slider(self,event):
    self.update_y_slice_index()

class MainFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, "3D Force Field Plot")
        pa_main = MainWindow(self)
        pa_main.szr_main.SetSizeHints(self)

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = MainFrame()
    frame.Show()
    app.MainLoop()
Appendix B - The AutoIt Script and Python API

#include <GuiListView.au3>
#include <GuiConstants.au3>
#include <GuiConstantsEx.au3>
#include <GuiImageList.au3>
#include <WindowsConstants.au3>
#include <GuiComboBox.au3>

; Include relevant UDFs. User Defined Functions, AutoIt ships with alot of these.
Global $title,$handle, $Path
Local $Listview, $ClassDetails
Local $Rowcount, $ColumnCount,$Row, $ColumnWidth, $ItemPos, $TotalWidth, $Column
Local $ComboView, $ComboList, $Instance, $ComboFocus

; Declare all my variables at the start instead of cluttering my functions with declarations.
$title = "Nanoscope"
$Path = "C:\Veeco\Z.exe"
$handle = WinGetHandle($title)

; If loop to see if we are calling the script with any parameters. $CMDLINE is a built-in AutoIt array used for investigating parameters, parameters are added to the $CMDLINE in the order
; they are written. For instance: *script* -parameter1 -parameter2 -parameter3 are added to place 1,2 and 3 in $CMDLINE. 0 is used to return the length of the array, recurring event in autoit
If not $CMDLINE[0] = 0 Then
    ; Run through all the parameters that were called with the script
    For $i = 0 To $CMDLINE[0]
        ; If one parameter is -EditItems call for EditItems() using the two following parameters as arguments
        If $CMDLINE[$i] = "-EditItems" Then
            EditItems($CMDLINE[$i+1],$CMDLINE[$i+2],$CMDLINE[$i+3],$CMDLINE[$i+4])
        ; If one parameter is -GetItem call for GetItem()
        ElseIf $CMDLINE[$i] = "-GetItem" Then
            GetItem($CMDLINE[$i+1],$CMDLINE[$i+2])
        ; If one parameter is -GetCurrentValue call GetCurrentValue() using the follow parameters as argument
        ElseIf $CMDLINE[$i] = "-GetCurrentValue" Then
            GetCurrentValue($CMDLINE[$i+1], $CMDLINE[$i+2], $CMDLINE[$i+3], $CMDLINE[$i+4])
    EndIf
EndIf
ElseIf $CMDLINE[$i] = "-GetCurrentValue" Then
GetCurrentValue($CMDLINE[$i+1],$CMDLINE[$i+2],$CMDLINE[$i+3])

; If one parameter is -GetComboBox call GetComboBox() using the following parameter
ElseIf $CMDLINE[$i] = "-GetComboBox" Then
GetComboBox($CMDLINE[$i+1],$CMDLINE[$i+2], $CMDLINE[$i+2])

; If one parameter is -EditComboBox call EditComboBox() using the two following parameters
ElseIf $CMDLINE[$i] = "-EditComboBox" Then
EditComboBox($CMDLINE[$i+1],$CMDLINE[$i+2],$CMDLINE[$i+3],$CMDLINE[$i+4])

; If one parameter is -Engage call Engage()
ElseIf $CMDLINE[$i] = "-Engage" Then
Engage()

; If one parameter is -Disengage call DisEngage()
ElseIf $CMDLINE[$i] = "-Disengage" Then
DisEngage()

; If one parameter is -Capture call Capture()
ElseIf $CMDLINE[$i] = "-Capture" Then
Capture()

EndIf
Next
EndIf

; Function to open Veeco
Func Open_Veeco()
Run($Path)
EndFunc
EndFunc

; A short explanation of the SysListView32 layout and how it is manipulated in AutoIt
;
;
; |Column0 | Column1 | Column2 | Column3 |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Item1</td>
<td>Height</td>
<td>0.300</td>
<td>Yes</td>
</tr>
<tr>
<td>--------</td>
<td>---------</td>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>Item2</td>
<td>Length</td>
<td>2.535</td>
<td>No</td>
</tr>
<tr>
<td>--------</td>
<td>---------</td>
<td>---------</td>
<td>---------</td>
</tr>
</tbody>
</table>
;
As we can see it very much resembles a table or a matrix where the Rows are called Items;
it makes it easier to understand the concept of SysListViews. An important fact to note;
UDFs only manipulate the first item and first column as a default. That’s why the

; Function for Selecting columns other than column0. Retrieves the position of the first
column using an UDF and adds the first columns width and all the following columns to
reach the chosen column. Backbone of practically all the functions following except

Func SelectColumn($handle,$hWnd, $Item,$Column)
For $i = 0 To $Column-1
$ColumnWidth = _GUICtrlListView_GetColumnWidth($hWnd,$i)
$TotalWidth += $ColumnWidth
Next
$ItemPos = _GUICtrlListView_GetItemPosition($hWnd,$Item)
ControlClick($handle,"","$hWnd","left",1,$TotalWidth + $ItemPos[0], $ItemPos[1])
EndFunc

; Function to expand all items in a ListView since items are hidden when contracted, which makes it impossible to retrieve information using the GetItem function.
; Clicks every item and presses -, loops through all items in a ListView.
Func ExpandItems($Instance)
WinActivate($handle)
WinWaitActive($handle)
$ClassDetails = "[CLASS:SysListView32; INSTANCE:" & $Instance & "; ID:1636; CLASSNN:SysListView32" & $Instance & "]"
$ListView = ControlGetHandle($handle,"", $ClassDetails)
$RowCount = _GUICtrlListView_GetItemCount($ListView)
$ColumnCount = _GUICtrlListView_GetColumnCount($ListView)
Local $s
$s = 0
While $s <= $RowCount
_GUICtrlListView_ClickItem($listview,$s)
Send("{NUMPADADD}")
$RowCount = _GUICtrlListView_GetItemCount($ListView)
$s = $s +1
WEnd
Send("!{tab}")
EndFunc

; Function that retrieves information about an specified item. This is stored in an array called $Row which contains the whole Row and every column is an index in $Row. There is a problem with
; the ItemTexts in the SysListView32, they are not used but just set to the default starting value, if someone manipulates a column manually in the veeco software this functions will return
; the wrong value. All of the functions in this script manipulate ItemText when changing values so GetItem will function properly.
Func GetItem($Instance,$Item)
WinActivate($handle)
WinWaitActive($handle)
$ClassDetails = "[CLASS:SysListView32; INSTANCE:" & $Instance & "; ID:1636; CLASSNN:SysListView32" & $Instance & "]"
$ListView = ControlGetHandle($handle,"", $ClassDetails)
$RowCount = _GUICtrlListView_GetItemCount($ListView)
$ColumnCount = _GUICtrlListView_GetColumnCount($ListView)
$Row = _GUICtrlListView_GetItemTextArray($Listview,$Item)
ConsoleWrite($Item & " ")
For $i = 1 To $ColumnCount
ConsoleWrite($Row[$i] & " ")

22
Next
ConsoleWrite(@LF)
256
EndFunc

; Function that circumvents our ItemText problem with GetItem. Highlights an area and
; to have GetItem use this instead of ItemText to return values.
Func GetCurrentValue($Instance,$Item, $Column)
WinActivate($handle)
WinWaitActive($handle)
$ClassDetails = "[CLASS:SysListView32; INSTANCE:" & $Instance & "; ID:1636; CLASSNN:SysListView32" & $Instance & "]"
$ListView = ControlGetHandle($handle,"", $ClassDetails)
SelectColumn($handle,$ListView,$Item,$Column)
Send("^c")
ConsoleWrite(ClipGet())
Send("!{tab}")
EndFunc

; Function for editing a specific position, utilizes SelectColumn. Selects the specified
Func EditItems($Instance,$Item,$Column,$msg)
WinActivate($handle)
WinWaitActive($handle)
$ClassDetails = "[CLASS:SysListView32; INSTANCE:" & $Instance & "; ID:1636; CLASSNN:SysListView32" & $Instance & "]"
$ListView = ControlGetHandle($handle,"", $ClassDetails)
$RowCount = _GUICtrlListView_GetItemCount($ListView)
$ColumnCount = _GUICtrlListView_GetColumnCount($ListView)
SelectColumn($handle,$Listview,$Item,$Column)
Send($msg)
Send("{ENTER}")
_GUICtrlListView_SetItemText($ListView,$Item,$msg,1)
Send("!{tab}")
EndFunc

; Function that retrieves all the combobox alternatives. Besides our regular EditBoxes
; the controller handle for the combobox to be able to use the UDF which we do using
; that can be toggled between two different values. Truth be told the SelectColumn is
; to at what position the combobox is located, the script doesn't detect what type of
Func GetComboBox($Instance,$Item,$Column)
WinActivate($handle)
WinWaitActive($handle)
$ClassDetails = "[CLASS:SysListView32; INSTANCE:" & $Instance & "; ID:1636; CLASSNN:SysListView32" & $Instance & "]"
$ComboBox = ControlGetHandle($handle,"", $ClassDetails)
$ListView = ControlGetHandle($handle,"", $ClassDetails)
SelectColumn($handle,$ListView,$Item,$Column)
$ComboFocus = ControlGetFocus($ListView)
$ComboView = ControlGetHandle($handle,"","[CLASS:ComboBox;" & " CLASSNN:" & $ComboFocus & "]")
$ComboList = _GUICtrlComboBox_GetListArray($ComboView)
For $i = 1 To $ComboList[0]
    ConsoleWrite($i-1 & " " & $ComboList[$i] & @LF)
Next
Send("!*{tab}"
EndFunc

; Same principle as GetComboBox, highlight the combobox and retrieve the controller handle.
; E.g: GetComboBox returns the following:
; 0 Hi
; 1 Bye
; 2 Hello
; Using EditComboBox with the index 2 would select "Hello" in our combobox’s dropdown list.
; the Combobox is located.
Func EditComboBox($Instance,$Item,$Column,$ComboItem)
WinActivate($handle)
WinWaitActive($handle)
$ClassDetails = "[CLASS:SysListView32; INSTANCE:" & $Instance & "; ID:1636; CLASSNN:SysListView32"
$Listview = ControlGetHandle($handle,"", $ClassDetails)
SelectColumn($handle,$Listview,$Item,$Column)
$ComboFocus = ControlGetFocus($ListView)
$ComboView = ControlGetHandle($handle,"", "[CLASS:ComboBox;" & " CLASSNN:" & $ComboFocus & "]")
_GUICtrlComboBox_ShowDropDown($ComboView,False)
$ComboList = _GUICtrlComboBox_GetListArray($ComboView)
_GUICtrlComboBox_SetCurSel($ComboView, $ComboItem)
_GUICtrlListView_SetItemText($Listview,$Item,$ComboList[$ComboItem],1)
Send("!*{tab}"
EndFunc

;Function to engage the cantilever. Activates our Nanoscope window and sends Ctrl+E
Func Engage()
WinActivate($handle)
WinWaitActive($handle)
Send("^e")
Send("!*{tab}"
EndFunc

;Function to disengage the cantilever. Activates our Nanoscope window and send Ctrl+W
Func DisEngage()
WinActivate($handle)
WinWaitActive($handle)
Send("\^w")
Send("!{tab}"")
EndFunc

;Function to capture. Send keyboard command to capture a file of the current scan. Haven't had the possibility to test this in the live environment. Might need some tweaking.
Func Capture()
WinActivate($handle)
WinWaitActive($handle)
Send("\^!c")
Send("!{tab}"")
EndFunc

#Import necessary addons
import os

Path = "VeecoV3.exe"

# Take note than the parameter Instance in the functions below tells the function which listview to operate on if multiple are in view.
# Function that expands all items in a listview
def ExpandItems(Instance):
os.popen(Path + " -ExpandItems " + str(Instance))

# Function that retrieves the name and value of every expanded item in a listview. They are stored in an array called Item
def GetItem(Instance, Item):
    Popen = os.popen(Path + " -GetItem " + str(Instance) + " " + str(Item))
    Item = Popen.readlines()
    print Item[0]

# Function that retrieves the current value in an item and returns it.
def GetCurrentValue(Instance, Item, Subitem):
    Popen = os.popen(Path + " -GetCurrentValue " + str(Instance) + " " + str(Item) + " " + str(Subitem))
    Item = Popen.readlines()
    print Item[0]

# Function that edits the item with the new values given as strings
# Item dictates rows in a listview and Subitem columns
def EditItems(Instance, Item, SubItem, Value):
os.popen(Path + " -EditItems " + str(Instance) + " " + str(Item) + " " + str(SubItem) + " " + str(Value))
# Function that retrieves all combobox alternatives of from the combobox given as a parameter
# Item dictates rows in a listview and Subitem columns
def GetComboBox(Instance, Item, SubItem):
    Popen = os.popen(Path + " -GetComboBox " + str(Instance) + " " + str(Item) + " " + str(SubItem))
    Item = Popen.readlines()
    for i in Item:
        print i

# Function the changes combocox value of a specific combobox. The Item, Subitem and Value are given as strings.
# Item dictates rows in a listview and Subitem columns
# Value begins at 1 instead of 0
def EditComboBox(Instance, Item, SubItem, Value):
    Popen = os.popen(Path + " -EditComboBox " + str(Instance) + " " + str(Item) + " " + str(SubItem) + " " + str(Value))
    Item = Popen.readlines()
    for i in Item:
        print i

# Function to engage the cantilever
def Engage():
    os.popen(Path + " -Engage")

# Function to disengage the cantilever
def Disengage():
    os.popen(Path + " -Disengage")

# Function to capture a picture
def Capture():
    os.popen(Path + " -Capture")