# A High-Level Nanomanipulation Control Framework

D. J. Arbuckle, J. Kelly and A. A. G. Requicha
(darbuckl | jonathsk | requicha@usc.edu)
Laboratory for Molecular Robotics
University of Southern California
*Los Angeles, CA, USA*

***Abstract*—Control systems for Atomic Force Microscopes (AFMs) tend to be specific to a particular model of device, and further have a tendency to require that they be written to target an inconvenient execution environment. This paper addresses these problems by describing a high-level programming system for an AFM in which the device-specific low level code has been separated into a different process accessible across the network. This frees the bulk of the code from the assorted constraints imposed by the specific device, and also allows for the insertion of an abstraction layer between the high level control code and the device itself, making it possible to write device independent control code.**

***Index Terms*—Nanomanipulation, Atomic Force Microscopy, High Level Control**

## I. Introduction

THERE is a great deal of potential benefit to be gained from the controlled structuring of matter on the nanoscale. Unfortunately, there are few techniques known for achieving the level of precision that we desire. One of the techniques that can achieve the desired precision is direct manipulation of matter by pushing it with the tip of an Atomic Force Microscope (AFM). For example, by pushing nanoparticles with the AFM tip, it is possible to place them with positioning errors of 1 nanometer or less.

Achieving such a degree of accuracy is no simple task, though. When operating on that scale, an AFM is a relatively clumsy manipulator, as thermal drift, actuator creep , hysteresis effects and other non-ideal behaviors all have a tendency to decrease positioning accuracy. Add to that the fact that an AFM tip is a single, somewhat round “finger” and that it is both the means of sensing and the means of actuation, but typically not both at once, and it is easy to see that controlling the AFM for manipulation is a difficult endeavor. The space of operating modes of AFM manipulation is a large one, and not yet fully explored, and so the correct way to achieve a given manipulation task is not always known. Finally, it is usually difficult to predict which facilities will be useful in a research environment, and a flexible development system that will encourage experimentation is a must. For all of these reasons, we need an AFM control framework which allows us to quickly develop and deploy AFM control code; that framework is what this paper describes.

This research was supported in part by NSF grant DMI-02-09678.

## II. Architecture

The software is broken into two components, a client and a server, which communicate with each other over TCP/IP. The client, which constitutes the majority of the program, is written in the agile language Python. This makes the development process quick and easy, particularly when creating and testing new control modules. The server could be written in any environment that supported networking and direct control of an AFM, but the one we have implemented is written in C++ and 16-bit Windows, targeting the Park Scientific (now Veeco) AutoProbe CP-R microscope. A block diagram of the complete system appears in Figure 1.

The client consists of a graphical user interface, a management layer that implements the AFM control logic, and a pair of threads which are responsible for performing network I/O. The management layer is made up of control modules. Control modules are composable, and so they can be used and re-used as building blocks for the construction of more complex control modules. The next section discusses control modules in more detail.

The server should do as little as possible. Its task is to receive the client's commands and do what it must to cause the AFM to execute them, and no more. This usually entails high-speed message handling, encoding and decoding data, and possibly some real-time processing. The server implements a small set of primitives for use by the client, and hides from the programmer the complexities of the digital signal processor code and other low-level aspects of the system.

The reason for splitting the client and the server across the network is that this decoupling of components allows the server to be written in whatever environment the AFM necessitates, but leaves the client free to operate in a more convenient and capable environment.

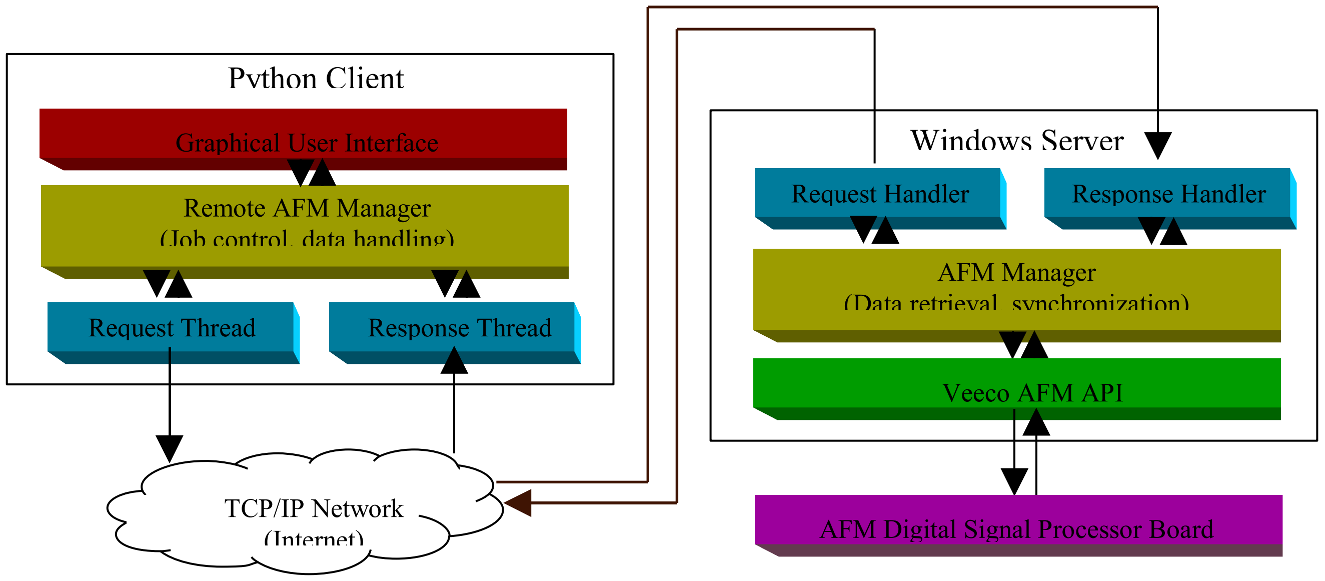

*Fig. 1 – Architecture of the AFM programming system*

## III. The Server

Much of the complexity in our server implementation is due to the necessity of targeting the Win16 Application Programming Interface. This necessitates a single-threaded event-driven architecture, using the Windows message queue for communication and synchronization with the AFM's digital signal processor driver. Internally, the server process maintains a state machine, driven by events pulled from the Windows message queue and from the network. This state machine manages executing and interleaving the assorted server-side operations.

The server's primary duty is to act as a thin wrapper around the AutoProbe Application Programming Interface. As such it spends most of its active time either receiving commands from the network, unpacking them, and invoking the AutoProbe API, or the reverse operation of receiving data from the AutoProbe API, packing them for transmission across the network, and transmitting them.

The AutoProbe Application Programming Interface is quite general at the conceptual level, leading us to suspect that it is similar to what a universal AFM API would look like. The development of a set of primitive operations that constitute a useful universal AFM API is a potential direction for future research.

## IV. The Client: Control Modules

The program provides a framework for the quick creation of AFM control modules. Several features are provided to make this a quick and easy task, including the ability to execute AFM operations either in parallel or in sequence and the ability to invoke other control modules in a nested fashion. Even though the underlying system is a collection of event handlers, the control module framework allows the controllers to be written in the more convenient paradigm of a sequence of imperative instructions with branches and loops.

For example, see Listing 1, which contains the complete code of a simple control module. This particular module repeatedly invokes a second module, so long as a flag is set. Note that the repetition is coded as a simple WHILE loop, even though it involves sending commands across the network to the server, and waiting for the server to respond. There is a separate thread that handles the actual communication, allowing control modules such as this one to block until it is completed. It is also noteworthy that the control module class to be invoked is a parameter of LibRepeatingWave. The availability of classes as first-class objects is one of the high-level features that makes Python well suited to our needs for rapid and flexible development.

For those not familiar with the Python programming language, Listing 1 describes a single class called LibRepeatingWave, which inherits from LibBase. LibRepeatingWave has the two methods required of any AFM control module: a constructor (*__init__*) and a main body (*execute*). The constructor consists of boilerplate code which varies little from one module to the next. The control module framework is such that invoking the constructor will eventually result in the main body being executed, either in the same thread as the constructor or in a new thread, depending on the value of the *spawn* parameter, which defaults to true. The *self* parameter is equivalent to Java or C++ *this*, and block structure is indicated by the degree of indentation.

Some of the more interesting control modules that we have written are compensators for actuator creep, hysteresis and thermal drift, modules for rectangular and single-line scans, and building on top of those, modules for locating the center of a particle and automatically pushing it to a desired location.

The drift compensation module cancels out the effects of thermal drift between the AFM tip and the surface beneath it. This is done by occasionally measuring the difference between scans taken in what should be the same place, and using a Kalman filter to track that offset. Then, when other modules command AFM operations, the requested coordinates of those operations are modified by the Kalman filter's current estimated value of the drift. This works quite well, and is necessary in order to have any sort of precision in fine nanoscale manipulation. This technique is discussed in detail in [1, 2].

Actuator creep and hysteresis are compensated for by another control module [3]. This module translates commands that would be well-suited to an ideal AFM, experiencing no creep or hysteresis effects, into more complex commands which achieve the desired results while negating creep and hysteresis.

The modules which perform rectangular or linear scans of the AFM tip over the surface have the ability to do so by way of the compensation modules, producing results which are both accurate enough and consistent enough over time to allow the scan data to be used as input to higher-level planning software.

Building on top of the scanning modules are controllers for high-level tasks, such as locating the center of a nanoparticle, moving a nanoparticle to some desired position, or automatically generating desired patterns of nanoparticles from initially randomly distributed particles deposited on a substrate surface.

Particle centers are located by repeatedly scanning the AFM tip over the general area occupied by a particle. Each scan is taken perpendicular to the previous scan, and crosses the path of the previous scan at the location where the maximum height was observed during that scan. Each repetition of this hill-climbing behavior provides a better approximation of the location of the true center of a particle, so long as the particle is not excessively irregular. The center-finding control module can perform this operation a fixed number of times, or until the change in the estimate between iterations drops beneath a threshold.

Particles can be pushed from one location to another by the AFM tip. The general details of this technique have been discussed in [4] and its references, as well as in [1]. In short, the particle is located using the center finding control module, a line between the actual particle location and the desired particle location is calculated, the feedback which prevents the AFM tip from impacting with the objects it is scanning over is disabled, and the tip is moved along the calculated line. During this operation the compensators are kept active, allowing the operation to proceed without the need to explicitly take the prominent sources of AFM positioning noise into account.

### *A.Application: automated construction*

The problem of automatically transforming a random distribution of particles into a specific predefined pattern is one we are still exploring. Our current approach is as follows:

First, use the compensated scan module to get an initial picture of the working area and estimates of the positions of the particles in it. The particle positions that can be read from that scan are not sufficiently precise for pushing, in spite of being compensated, because each pixel of scan data represents too large an area of the surface. That being the case, the estimated particle positions are refined by applying the center-finding mechanism.

```
from module_globals import *

class LibRepeatingWave(LibBase):
    def __init__(self,
                 manager,
                 initial_params,
                 module,
                 spawn = True,
                 notify = True):
        LibBase.__init__(self, manager, notify)
        self.useLock = False
        self.spawn   = spawn
        self.params = initial_params
        self.params.repeatFlag = False

        if spawn:
            self.thread.start()
        else:
            self.run()

    def execute(self):
        try:
            LibBase.execute(self)
            manager = self.manager
            manager.begin(self.useLock)

            while not self.terminateFlag:
                self.params.repeatFlag = False
                module(manager, self.params,
                       False, False)

            manager.end(self.notify)
        finally:
            if self.spawn:
                thread.exit()
```

*Listing 1: A simple AFM Control Module*

Once the particle centers have been determined with sufficient accuracy, the Hungarian Method [5, 6] is used to assign specific particles to occupy specific positions in the target pattern. This quickly makes a good assignment of particles to positions, but it doesn't take into account that there may be more particles within the target area than are necessary, and those excess particles have to be removed. Thus, after the assignment is made, unassigned particles are counted, and an equal number of new target positions are added in the area outside of the actual goal shape. A new iteration of the Hungarian Method is then used to assign particles to the targets, with this iteration considering only those particles that were assigned to target in the previous pass or which are within the target area.

That assignment of particles to positions takes no account of the possibility that actually placing a particle into a position might require that it move through one or more other particles, which is impossible. For this reason, each planned pushing path is check for collisions. If there are collisions, the particle to position assignment that produced the collision is replaced by a pair of new assignments; the particle that would be collided with is reassigned to the target of the original assignment, and the particle that the original assignment specified us reassigned to the position occupied by the interfering particle. Once all such reassignments have been performed, the set of particle assignments constitute a plan for pushing particles in order to construct the target pattern.

Finally, once the plan is determined, it is used to drive the AFM tip, using the compensation, the single-line scan, and the pushing control modules.

## V. In Detail: Scanning along a line

In this section, we present an example action, and follow it in detail as it flows through each part of the system. The action in question is to scan the AFM tip across the surface beneath it, collecting the topographical data that it produces.

First comes the invocation of an AFM control module on the client; we'll call it LibSingleLine in this discussion. The purpose of LibSingleLine is to control exactly the action we are interested in: scan the AFM tip along a single sweep across a surface, and record the data so produced. Depending on the *spawn* parameter of the LibSingleLine constructor, the control module may execute in either the calling thread or in a thread of its own; we'll assume for this example that *spawn* has a value of *false*, and so the module will run in the same thread that invoked it. This ability to run control modules in either serial or parallel mode as needed has proven to be very convenient.

The LibSingleLine constructor was also parameterized with an assortment of values, notably the coordinates between which we want to sweep the AFM tip. These coordinates are provided in a global reference frame. If the drift compensator is not running, this whole frame slowly moves relative to the surface. We'll assume that the drift compensator is running, though, and so the input reference frame can be transformed into the surface-affixed reference frame fairly easily. Since rotation between the frames is effectively precluded by the nature of the hardware, the transformation is just a matter of translation. The drift compensator, which is a singleton control module, maintains the current translation in an accessible location, and it is simple added to the start and end coordinates by vector summation.

In order to achieve maximum precision, the LibSingleLine module must also employ the creep and hysteresis compensator. This compensator works by adjusting in detail the driving waveforms that are applied to the AFM's actuators, and so it can not be run until after the gross transformation of drift compensation has already been applied. The desired endpoints of the scan, as transformed by the drift compensation, are passed to the creep and hysteresis compensator, which returns the correct driving waveforms.

Having completed the assorted preprocessing steps, including the above described applications of the compensators as well as generating a few other simpler control waveforms, LibSingleLine invokes the client-side remote procedure call stub for the server's interface, passing it the calculated control waveforms. The thread executing is then blocked pending the return of an error code from the server. Once the error code is received, if it does not signify an error, the thread is again blocked until the requested data arrive.

The stub packages the request for transmission to the server, using a wire protocol implemented on top of XML-RPC [7]. The request package is transmitted to the server via normal Internet protocols, and the server sends back an error code through the same medium. The control module's thread is unblocked and the error code is passed to the control module, which then determines whether to block the thread again to wait for data. We'll assume that everything went well and that the control module re-blocked its thread.

On the server side, the server receives the request package and unpackages it, then tries to interpret it as a call to the underlaying API. If it can't figure out what call the package represents, an error code is sent to the client and the server goes back to waiting for requests.

In our example, the server recognizes the packet as representing a request for a single sweep of the AFM tip, and invokes the API function to perform that task. That API function produces an error code or a job handle. In our example we'll assume that a job handle is produced, which we will transform into the “no error” code, and that error code is then transmitted to the client, and eventually all the way up the stack to the control module, which uses it to decide whether to block waiting for data.

The API, in our case, is a 16-bit Windows library and an associated driver for communicating with the digital signal processor in the AFM. These two components of the API communicate with each other via the Windows message queue, which produces a somewhat unusual architecture by more modern standards. The end result is that, for the duration of the API call, the server must be listening to the Windows message queue for messages indicating the receipt of data, and periodically polling the API for those data as well. This inconvenient system is one of the major sources of impetus in our efforts to separate high-level and low-level AFM control code.

Eventually, the data are acquired, either from a queued message or from a periodic poll. At this point the server packages them up, again using an XML-RPC wire format, and transmits them to the client. The client unpackages them and makes them available to the control modules, and LibSingleLine unblocks. Finally LibSingleLine makes the data available as its result, and terminates execution. It is only possible to have one data-producing operation executing on the server at a given time, so there is no ambiguity regarding which control module should handle the data, in spite of the fact that many control modules can be executing on the client side at any given time.

## VI. Conclusion

We have constructed a software environment that facilitates experimentation with, and automation of, the controlled manipulation of nanoparticles and similarly-scaled nanostructures. This environment presents to the programmer a comfortable and rapid way of creating and modifying AFM control code. This environment allows the programmer to write the code at progressively higher levels of abstraction, building control modules by composing lower-level control modules. We have also written a number of such control modules, which boost the level of abstraction up to the point where the common sources of noise in the positioning of the AFM tip can be ignored, and where in fact an operation such as "move particle X from location Y to location Z" can be thought of as a single reliable primitive.

This software environment allows rapid progress in the development of new nanomanipulation techniques and results. Although nanomanipulation was our primary motivation for system development, the facilities we provide are equally useful for other tasks such as AFM nanolithography.

## References

[1] B. Mokaberi and A. A. G. Requicha, "Towards automatic nanomanipulation: drift compensation in scanning probe microscopy", *Proc. IEEE Int'l Conf. on Robotics & Automation (ICRA '04)*, New Orleans, LA, pp. 416-421, April 25-30, 2004.

[2] B. Mokaberi and A. A. G. Requicha, "Drift compensation for automatic nanomanipulation with scanning probe microscopes", *IEEE Trans. on Automation Science & Engineering*, Vol. 3, No. 3, pp. 199-207, July 2006.

[3] B. Mokaberi and A. A. G. Requicha, "Compensation of scanner creep and hysteresis for AFM nanomanipulation", accepted for publication, *IEEE Trans. on Automation Science & Engineering*, 2006.

[4] A. A. G. Requicha, S. Meltzer, F. P. Teran Arce, J. H. Makaliwe, H. Siken, S. Hsieh, D. Lewis, B. E. Koel and M. Thompson, "Manipulation of nanoscale components with the AFM: principles and applications", *IEEE Int'l Conf. on Nanotechnology*, Maui, HI, October 28-30, 2001

[5] Harold W. Kuhn, "The Hungarian Method for the assignment problem", *Naval Research Logistic Quarterly*, **2**:83-97, 1955

[6] J. Munkres, "Algorithms for the Assignment and Transportation Problems", *Journal of the Society of Industrial and Applied Mathematics*, **5**(1):32-38, 1957 March

[7] Dave Winer, "XML-RPC Specification", *http://www.xmlrpc.com/spec*, June 2003