QDG Python Controller and PAT Framework Manual

Victor I. Barua

August 28, 2013

Distillation of four months of programming work during the summer of 2013. If question arise they can be directed to *victor.barua@gmail.com*. All of the code written during this work term can be found here:

`https://github.com/vbarua/QDG-Lab-Framework`

Contents

1 Device Controllers

The main component of each controller module is the controller class which encapsulates all of the attributes and functions that the device associated with it can carry out. The controllers can be used in other Python scripts by importing the controller class from the module, creating an instance of it and then calling the required commands through that instance. Every controller is capable of at least taking data and saving it to an external file. Additional capabilities like automated processing and plotting of data have also been implemented in some cases. Each controller module also has a threading class which is used in conjunction with the controller class to allow for data to be taken without blocking the main thread. These controllers have been designed so that all of their data collection parameters (ie. sample rate, collection duration, etc) are set when they are constructed. Following this a simple method call, usually something like takeData(), can be used to take data with the controllers internal settings.

The controller documentation in this text gives a broad overview of each controller. The modules themselves have been written with ease of readability in mind and should in-theory be relatively self-documenting. Each controller module contains example code that will take data when called directly from the command prompt. This code is at the bottom of each module and serves as an example of how data can be taken. Manuals for the controllers can also be found in the DeviceControllers folder of the PAT Framework.

1.1 Lab Jack

1.1.1 Controller Parameters

- activeChannels: Array of size 1,2 or 4 indicating which channels to collect data from.
- sampleRatePerChannel: Number of samples taken per channel per second.
- scanDuration: Time over which to collect data (in seconds).
- trigger: Determines whether a trigger will be used.
- triggerChannel: Digital IO channel to use as trigger.
- idnum: Local ID of LabJack to utilize. -1 uses first available LabJack.

1.1.2 Software Installation

The software required to utilize the Lab Jack U12 can be found here: `http://labjack.com/support/u12`. This will install the necessary drivers the device controller utilizes (specifically the ljackuw.dll library).

1.1.3 Device Details

The Lab Jack is a USB data acquisition device that can both measure and output digital and analog voltage signals. Based on the needs of the QDG lab, only its analog voltage capture capabilities have been implemented. Using the Lab Jack in single-ended mode, either 1, 2 or 4 analog channels (AI in figure 1) can be read simultaneously with an aggregate sample rate between 200-1200 samples per second. Each channel has 12 bits of resolution over a range of $\pm$10V.

1.1.4 Controller Details

When a LabJackController is created, it connects to the first available Lab Jack and configures it using the settings passed to the constructor. Data is taken by simply calling collectData(), after which a call to save() can be used to save the data. This method of taking data will however prevent you from running other code whilst data is being collected. If you wish to run other code, instead of calling collectData() call start(). Then before calling save() call end().

The Lab Jack can also be utilized with a software trigger by using the appropriate configuration. By virtue of it being a software trigger however, delays on the order of milliseconds can be expected. If triggering functionality is required, it would be better to used the PMD device.

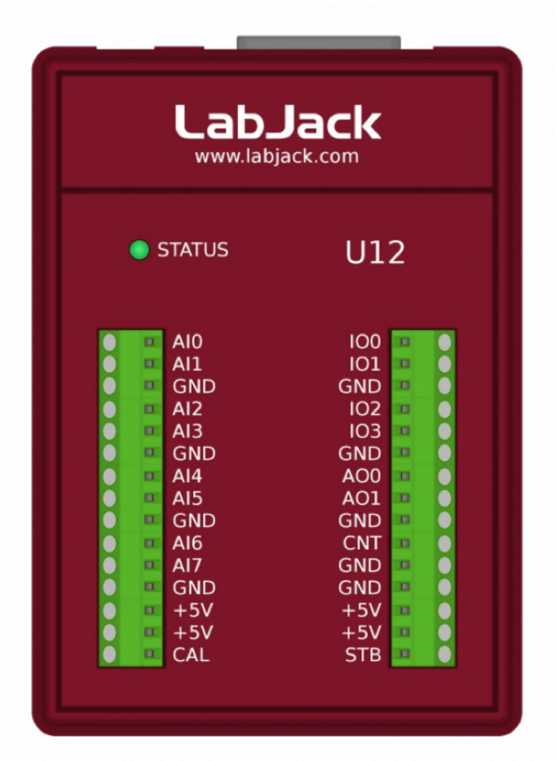

Figure 1: Lab Jack U12

1.2 Personal Measurement Device (PMD)

1.2.1 Controller Parameters

- activeChannels: Array of channels from which to record data.
- gainSettings: Array of gain settings for each of the channels. Must have same size as activeChannels.
- sampleRatePerChannel: Sample rate per channel.
- scanDuration: Duration of scan in seconds.
- vRange: Voltage range identifier code from PMDTypes file.
- trigger: Boolean to indicate whether a trigger should be used or not.
- trigType: Trigger type identifier code from PMDTypes.
- boardNum: Board number registered through InstaCal program. 0 will use first available board.

1.2.2 Software Installation

The software required to utilize the PMD-1208FS can be found here: `http://www.mccdaq.com/software.aspx`. Specifically, download and run the MCC DAQCD package. The device controller makes use of either the cbw32.dll or cbw64.dll depending on the bitness of the OS (ie. 32 or 64 bit). Make sure that the right version is set in the PMDController file, otherwise an error will be raised when attempting to run PMD code.

Before data can be taken, it is necessary to run the InstaCal program (it should have been installed with the previous package) with the PMD connected to the computer. This program will detect a device and create a configuration file which is needed to utilize it.

Figure 2: PMD-1208FS

1.2.3 Device Details

The PMD-1208FS (Figure 2) is a USB data acquisition devices that can both measure and output digital and analog voltage signals. Based on the needs of the QDG lab however, only its analog voltage capture capabilities have been implemented. Using the PMD in differential mode, up to 4 channels can be read at a maximum theoretical aggregate rate of 50kHz, though this rate will depend on the capabilities of the computer it is connected to.

In order to measure a voltage with the PMD, connect the signal to one of the "*CHx IN HI*" pins, the signal ground to an "*AGND*" pin, and finally the corresponding "*CHx IN LOW*" to the same "*AGND*" as before. The pin diagram can be seen in Figure 3. In differential mode, the PMD provides 12 bit resolution in its voltage measurements. The voltage range can be set to any of the range values listed in the PMDTypes file. This file holds constants for ease of access. It is recommended that the smallest range that includes the signal to be measured be utilized in order to maximise the resolution of the measurements.

The PMD can also be utilized with a hardware trigger by using the appropriate configuration. The trigger input should be connected to Channel 18 (TRIG IN in Figure 3). By default the PMD will trigger on a rising edge (though triggering on a falling edge is also available).

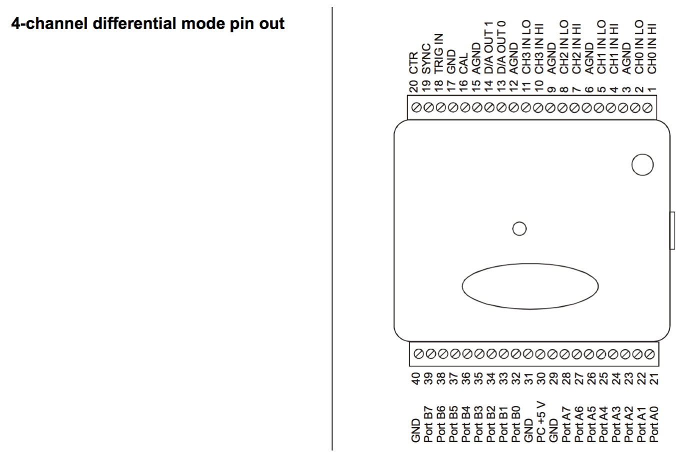

Figure 3: Pin layout for differential mode.

1.2.4 Controller Details

When a PMD Controller is created, it connects to the first available PMD and configures it using the settings passed to the constructor. Data is taken by simply calling collectData(). After this calling processData() will extract the readings from the data buffer and a subsequent call to save() will save the data. This method of taking data will however prevent you from running other code whilst data is being collected. If you wish to run other code, instead of calling collectData() and processData() call start(), Then before calling save() call stop().

1.3 Stabil Ion Controller

1.3.1 Controller Parameters

- port: COM port to which gauge is connected.
- duration_s: duration of data collection in seconds.
- secondsPerSample: the number of seconds to wait between samples. Not recommended to be anything less than 1.

1.3.2 Software Installation

The Granville-Phillips Series 370 Stabil-Ion Gauge Vacuum Gauge is controlled via the RS232 communication protocol. The pySerial package found here `http://pyserial.sourceforge.net/` can be used to communicate with the gauge.

1.3.3 Controller Details

The Stabil Ion controller subclasses the Serial class (from pySerial) in order to have access to its services. To create an instance of the controller, it is necessary to know the COM port to which it is connected. A list of COM ports is available within the device list on Windows. Note that communication ports are listed using one-based indexing, which means that COM2 would correspond to port 1.

The majority of this controller is wrapper for RS232 commands. It depends heavily on the read and write methods provided by the Serial class. write() is used to send message strings to the device, and read() is used to read messaged returned from the device.

The RS232 protocol for this device indicates that all commands must be terminated by a carriage-return and line-feed. For this reason, the write() method has been overwritten to append "\r\n" to all messages sent. read() has also been overwritten to remove these two characters from the returned data.

To collect data with the controller, simply call the collectData() method. The controller will then request a reading from the gauge every time a reading is required. Note that as the timing of readings is being controlled via software, time resolution cannot be guaranteed beyond 1s. After data is collected, it can be saved to a csv file by calling saveData().

1.4 MKS SRG3

1.4.1 Controller Parameters

- port: COM port to which gauge is connected.
- duration_s: duration of data collection in seconds.
- measurementTime_s: time over which a single measurement will be made.
- gType: the type of gas being measured. See controller file for the numeric codes.
- pUnits: the pressure units used for outputs. See controller file for the numeric codes.
- tUnits: the temperature units used for inputs. See controller file for the numeric codes.

1.4.2 Software Installation

The MKS SRG3 spinning rotary gauge is controlled via the RS232 communication protocol. The pySerial package found here `http://pyserial.sourceforge.net/` can be used to communicate with the gauge.

1.4.3 Controller Details

The MKS SRG3 controller subclasses the Serial class (from pySerial) in order to have access to its services. To create an instance of the controller, it is necessary to know the COM port to which it is connected. A list of COM ports is available within the device list on Windows. Note that communication ports are listed using one-based indexing, which means that COM2 would correspond to port 1.

The majority of this controller is wrapper for RS232 commands. It depends heavily on the read and write methods provided by the Serial class. write() is used to send message strings to the device, and read() is used to read messaged returned from the device.

The RS232 protocol for this device indicates that all commands must be terminated by a carriage-return. For this reason, the write() method has been overwritten to append "\r" to all messages sent. read() has also been overwritten to remove this character from the returned data.

To collect data with the controller, simply call the collectData() method. The controller will then take a reading from the gauge whenever one becomes available, which can be anywhere from 5 to 60 seconds. After data is collected, it can be saved to a csv file by calling saveData()

1.5 Point Grey

1.5.1 Controller Parameters

- numOfImages: Number of images which will be taken.
- expTime_ms: Exposure time for each image in milliseconds.
- gain: Camera gain level.
- roi: Region of Interest object. If false the full region will be utilized, otherwise the specified region of interest will be used.
- speedBoost: When toggled the camera will be able to operate at a higher frame rate but images will have a lower bit depth.

1.5.2 Software Installation

The controller requires various drivers to operate correctly. The required drivers can be obtained here: `http://www.ptgrey.com/support/downloads/downloads_admin/Index.aspx`. Note that the site requires a login. After logging in, select the software tab and download the FlyCapture v2.4 Release executable. The installation will install all of the necessary drivers. It will also install the Point Grey FlyCap2 utility which provides a useful interface for viewing the camera status and configuration.

1.5.3 Device Details

The Point Grey controller was written and tested specifically for the Flea2 camera (model FL2G-13S2M-C). The particular camera used in the lab can operate in either Y8 or Y16 mode, which correspond to monochrome images with either 8 or 12 bits of depth per pixel. The full 16 bits of Y16 mode are not available due to limitations of the A/D converter. The Flea2 has a maximum resolution of 1280x960, and depending on whether it is in Y8 or Y16 mode a frame rate of either 15 or 30 fps respectively. However by using the region of interest settings to reduce the data collection area it is possible to exceed these frame rates. This is important because the camera cannot be triggered consecutively at a rate faster than its frame rate.

Figure 4: Flea2 Camera

1.5.4 Controller Details

When a Point Grey controller is created, it connects to the first available camera and configures it using the settings passed to the constructor. Hardware or software triggering can be enabled by calling the corresponding enableSoftwareTrigger() or enableHardwareTrigger() methods, but they are not necessary. The software trigger can be fired by using the fireSoftwareTrigger(). Before calling the start() method, it is necessary to call the setDataBuffers() method which tells the camera how many triggers to expect and/or images to take.

At this point the start() method which will make the camera wait for triggers can be called. After all of the triggers have been recieved the stop() method should be called, after which data can be saved by calling any combination of the saveRAWImages(), savePNGImages() and saveLog() methods. The last one of these saves a text log of the camera settings used for the collection as well as the time at which each image was taken compared to the first image.

1.6 PixeLink

The PixeLink controller was written by Ovidiu Toader. It has been interfaced into the PAT Controller via the PixeLinkMediator, however it requires a number of device specific commands to function properly. The documentation here specifies its usage within the PAT Framework.

1.6.1 Controller Parameters

- gain: Camera gain level.
- expTime_ms: Exposure time for each image in milliseconds.
- ROI_width: Width of the region of interest in pixels.
- ROI_height: Height of the region of interest in pixels.
- ROI_left: Offset from the left of the region of interest in pixels.
- ROI_top: Offset from the top of the region of interest in pixels.
- useROICenter: Boolean indicating whether the region of interest should be centered using the ROI_center value.

- ROI_center: Tuple indicating the location of the center of the region of interest.

1.6.2 Device Specific Function

The PixeLink has three functions unique to it in the PATController class (more on this later). The first is triggerPixeLink() which will send a trigger pulse via the UTBus to the camera and also increment an internal trigger counter. This counter is used by the second function, which is setPixeLinkImageCount(). This function should be called before startDevices() and after all of the triggerPixeLink() calls, and sends a message to the server notifying it of how many triggers to expect. The third function is flushPixeLink(), which should be called whenever the camera fails to collect data. The notification for this arrives via the stopDevices() function in the PATController. The PixeLinkTest script gives an example of how to use the camera within the framework.

2 PAT Control System - General Usage

The PAT control system draws heavily from the Python framework developed for the MOL experiment. The main component of the system is the PATController class which extends the Recipe class from the UTBus. This means that everything that can be done with the Recipe can also be done with the PATController. The power of the PATController comes from the fact that it consolidates all of the functionality available to the PAT experiment in one class. Using the PATController requires that an instance of the PATServer be running on the machine collecting data.

2.1 Script Breakdown

What follows is a breakdown of a script for controlling the PAT system. First the PATController is imported along with the defaultSettings of the system and the overwriteSettings() function. The next import is an updatePackage (more details on this later) from the updateSettings files.

```
from PATController import PATController, defaultSettings,
    overwriteSettings
from updatedSettings import updatePackage
```

Using the overwriteSettings() function, the updatePackage can be used to overwrite settings in the defaultSettings. These updated settings are then used to construct a PATController object (named PATCtrl in this example).

```
updatedSettings = overwriteSettings(defaultSettings, updatePackage)
PATCtrl = PATController('Example', updatedSettings)
```

Next the start() method is called which allows UTBus commands to be recorded.

```
PATCtrl.start() # Starts recording UTBus Commands
```

Following this, a series of UTBus commands defined directly in the PATController can be called.

```
PATCtrl.set_2D_I_1(3.9)
PATCtrl.close_all_shutters()
PATCtrl.set_3D_coils_I(1.2)
PATCtrl.set_3DRb_pump_amplitude(0.8)
PATCtrl.set_3DRb_pump_detuning(12)
PATCtrl.wait_s(5)
PATCtrl.open_all_shutters()
```

Note that the UTBus commands won't be executed until the end() method is called, which will take the recorded UTBus commands, compile them and send them to the UTBus. The end() method also blocks the script until the UTBus finished. For this reason, if data is to be taken the startDevices() function should be called before it. This can be seen below.

```
PATCtrl.startDevices()  # Start recording devices.
PATCtrl.end() # Stop recording UTBus commands and execute them.
```

Before saving data it is necessary to call the stopDevices() function which, depending on the device, will either stop data collection or wait until the device is finished collecting data.

```
PATCtrl.stopDevices()   # Stop recording devices.
PATCtrl.save()          # Save data from devices.
```

Once the script is done the closeClient method should be called. This notifies the server that the script is done.

```
PATCtrl.closeClient()   # Close link between controller and server.
```

2.2 Example Scripts

Along with this document, the QDG-GATEWAY machine contains a folder of example scripts that showcase some of the capabilities of the framework.

- VariableDataCollectionParamaters: This script showcases how to change data collection parameters between trials.
- RunMultiple: This script runs and processes data from multiple devices simultaneously.
- FluorescenceOptimization: Showcases the optimization capabilities of the framework.
- TestScripts: Folder of scripts to collect data individually with each device.

3 PAT Control System - Implementation Details

The PAT control system has been split into two main components: a client and a server. The client side is setup on the QDG-GATEWAY machine and interfaces with the UTBUS. The server side is setup on the QDG-PATPC machine and interfaces with all of the data collection devices. Originally there was no such separation and everything was running on the QDG-GATEWAY machine, however as data collection is a processor intensive task and this is a shared machine this proved problematic.

3.1 Devices

3.1.1 DeviceControllers

The DeviceControllers folder contains the individual controllers for each of the data acquisition devices available to the PAT system. These controllers are the same as those described in the Controller section and in fact can be used independently of the control system.

3.1.2 DeviceMediators

The DeviceMediators folder contains a mediator module for each of the controllers available to the PAT system. Each of the mediators must implement the methods of the DeviceMediatorInterface class, which mandates common methods like start() and save(). This means that anyone writing a device controller doesn't have to worry about how it will plug into the framework as long as general capabilities likes starting data collection and saving data are available.

3.2 Settings

3.2.1 DefaultSettings

The DefaultSettings folder contains settings files which represent the default settings of various components of the PAT system. In general these settings files should not be modified, rather any controller instance that needs to utilize them should import and overwrite them.

3.2.2 Settings Class

The Settings module defines a custom dictionary class which prevents new items from being added to the dictionary after it is created. This makes overwriting settings safer as invalid names won't be accepted.

3.2.3 SettingsConsolidator

The SettingsConsolidator module is used to pack up all of the individual settings files into a defaultSettings Setting dictionary which is used to construct a PATController object. The defaultSettings dictionary consists of two subcomponents, the deviceSettings dictionary which stores the default information associated with devices and the generalSettings dictionary which stores everything else. The first half of the module imports the individual settings files and then places them into either the deviceSettings or generalSettings dictionaries.

The second half of the module converts all of the regular dictionaries into Settings dictionaries and packs them into defaultSettings. It also defines the overwriteSettings function which can be used to overwrite the defaultSettings dictionary.

3.2.4 TemplateGenerator

The TemplateGenerator script takes all of the settings in the DefaultSettings folder and generates a file, settingsTemplate.py, with all of the possible settings that can be modified present.

3.3 Communication

3.3.1 Communication Protocol

Both the PATServer and PATClient are able to send and receive messages. Only one client can be connected to the server at any point in time. Messages are sent and received in two parts. First the sender sends a 4 character string to the receiver indicating the length of the message, after which the actual message is sent. This protocol ensures that messages arrive completely.

On the server, messages are parsed by the intepretMessage function. The first character of any message is a control character which is used by this function to dictate the servers response.

3.3.2 PATServer

The PAT Server is a socket server that listens for a connection by a PAT Client. When a client connects, the server will dedicate itself to the client until it disconnects meaning that no more than one client can connect at a time. On connection the server also creates all of the device mediators needed based on the initialization settings and stores them in a list, which allows for easy iteration over common device mediator methods like start() and save().

3.3.3 PATClient

The PAT Client can send messages to the PAT Server to execute the various methods of the device mediators. It can also send messages to execute functions from specific device mediators.

3.4 Save Controller

The SaveController module is used by the server to generate the file paths to which experimental data will be saved. In a multi-trial run, the state of the SaveController is saved after each trial. If at any point the server is interrupted or crashes, it is possible to reload the server state and continue the experimental run.

3.5 Device Mediator Interface

The Device Mediator Interface class defines a number of methods that all device mediators must implement. The core methods are the start(), stop() and save() methods. These must have proper implementations. All other methods can either be implemented or call the pass function.

3.5.1 start

The start method is used to signal a device to either start taking data or prepare for a trigger signal. Data collection must happen in a seperate thread.

3.5.2 stop

The stop method can either stop a device or wait for it to finish collecting data.

3.5.3 save

The save method must take a folder path. The device must then save its data to the folder given.

3.6 Explanation of Common Settings Options

- takeData: Boolean that determines whether data is taken or not.
- processData: Boolean that determines whether data is processed when processData() is called.
- persistent: Boolean that determines whether a device should have a persistent state between trials. Utilizes the saveState() and restoreState() methods of the DeviceMediatorInterface.
- dataFolderName: The file name under which data taken by the device will be saved.

4 PAT Control System - Useful Tips

4.1 Starting the Server

The server can be started by running the PATServer file, which will start a new instance of the server. However it is also possible to restore the server to a particular save state from a multi-trial run so as to continue saving data in the same data file with the correct trial numbers. This was implemented mainly for usage with the optimizer built into the framework for cases in which the server crashed. When running a script that utilizes saveTrial(), the state of the saveController is saved in the experiment folder after each trial. If the server crashes and needs to be restored, running:

```
python .../PATServer .../ExperimentDataFolder
```

in the command prompt will restart the server in the state it was in before the crash. It is also possible to restore the state of the server to that after any particular trial by copying the SaveController.pkl file from the trial folder into the experiment folder.

4.2 Handling Data Collection Failures

During data collection, the PATServer keeps track of any devices that fail to take data. When stopDevices() is called on the client side by the PATController, it will return a boolean value that indicates whether any of the data collection devices has failed. This feature is useful for multi-trial runs to prevent holes in the data from unsuccessful trials. By checking the return value of this function and only saving data when it is false, it is possible to repeat a particular trial an arbitrary number of times until data is collected possibly. This status checking behaviour can be seen in the *PixeLinkTest* script and the *FluorescenceOptimizer* script.

4.3 Adding a New Controller

Adding a new controller to the PAT system requires modifications in a number of places.

1. Add the controller module to the *Server/DeviceMediators/DeviceController* folder. The controller must be capable of starting a new thread and collecting data in it.
2. Create a new mediator in the *Server/DeviceMediators* for the controller. The mediator should extend the DeviceMediatorInterface class.

3. Add an import statement for the controller in the PATServer file.
4. Add a default settings file for the controller in the *Client/Settings/DefaultSettings* folder.
5. Import the defaultSettings into the SettingConsolidator file (in *Client/Settings*) and add an entry into the deviceSettings dictionary.

4.4 Porting the Framework

Porting the framework to another one of the experimental systems requires a number of steps. First copy all of the framework files to the machine hosting the UTBus modules and the machine connected the data collection devices. Then:
Server Changes

1. Modify the Python environment path to include both the Client and Server folders.
2. In the PATServer file, change the PORT value to an open one on the machine and the dataPath value to the location in which data should be saved (make sure to create this folder if it doesn't already exist).
3. Remove the device controllers and mediators for devices which are not available to the setup.

Client Changes

1. Modify the Python environment path to include the Client folder.
2. Remove unnecessary settings from the DefaultSettings folder as well as from the SettingsConsolidator file.
3. Change the HOST and PORT in the PATClientSettings file to match that of the server.
4. Change the database used by the PATController within the __init__() function from 'PAT' to the appropriate one for the experiment.
5. Modify/replace the functions in the PATController to those the of the experiment module replacing it.

Afterwards, it is highly recommended that all references to PAT- be replace by YourExperimentDeviceHere-. This includes file names and contents.

5 Optimizer

5.1 Particle Swarm Optimization

Particle Swarm Optimization (PSO) is an iterative optimization method. Each particle in the optimization has a position and a velocity. The position represents the location of the particle in the parameter space, and the velocity describes how the particle traverses the space. The particles are also experience an "attraction" to the best global particle (the particle with the best fitness value so far out of every particle evaluated) and the best position that particle has had in its past. Every iteration, each particles fitness is evaluated and its historical best along with the global best are updated. Once every particle has been evaluated, new velocities are calculated for the particles, then the particles are moved to new positions based on their velocities and a new iteration can begin.

5.2 Initialization

When the optimizer first start, it will seed the parameter space with randomized particles that are initialized as follows:

$$\mathbf{s} = \mathbf{U}(\mathbf{s_{min}}, \mathbf{s_{max}})$$
$$\mathbf{v} = \mathbf{U}(\mathbf{v_{min}}, \mathbf{v_{max}})$$

where the min and max values represent the boundaries of the search space.

5.3 Update Formula

The velocities of the particles are updated based on the following formula.

$$\mathbf{v} \leftarrow \omega\mathbf{v} + \phi_p r_p(\mathbf{s_p} - \mathbf{s}) + \phi_g r_g(\mathbf{s_g} - \mathbf{s})$$

ω is a damping factor on the particles current velocity. ϕ_p and ϕ_g are tunable weighing factors towards the particles historical best and global best respectively. r_p and r_g are random values chosen from $U(0,1)$. $\mathbf{s}$ represent the particles position and $\mathbf{s_p} - \mathbf{s}$ and $\mathbf{s_g} - \mathbf{s}$ are vectors towards the particles historical best and the global best respectively. With the velocity determined, calculating the particles new position is simply:

$$\mathbf{s} \leftarrow \mathbf{s} + \mathbf{v}$$

5.4 Boundaries

As part of the optimization process, it is necessary to pass bounds for every parameter to the optimizer. These bounds are used as hard bounds on the particles position and also used to limit the particles max velocity based on the following formula:

$$|\mathbf{v}| \leq (\mathbf{s}_{max} - \mathbf{s}_{min})\,\alpha$$

Where α is a parameter between 0 and 1 that can be tuned to limit the maximum speed.

When a particle exceeds a boundary in any of its dimensions, the position of that particle will be readjusted as either

$$s \leftarrow U(s_{min}, s) \quad or \quad s \leftarrow U(s, s_{max})$$

depending on whether the lower or upper boundary was exceeded. The velocity will then be adjusted to:

$$v \leftarrow s - s_{old}$$

These adjustments are based on the PSO Bound Handling and PSO High-Dimensional Spaces papers found in the Manuals folder.

5.5 Implementation/Usage Details

The PAT framework has PSO built into it via the ParticleSwarmOptimizer module which is treated as just another device by the framework. Like the other devices, the optimizer has a settings dictionary associated with it. The following are unique settings it includes.

- paramBounds: A tuple of 2-tuples representing the lower and upper bounds for parameters.
- numOfParticles: The number of particles in the swarm.
- numOfGenerations: The number of generations to iterate the swarm over.
- fitnessEvalScript: A path to the fitness evaluation script on the server. This script will be executed in the current trial folder to evaluate the current particles fitness.
- phiG (ϕ_g): Weighing factor towards the global best.
- phiP (ϕ_p): Weighing factors towards a particles historical best.
- w (ω): Velocity damping factor.
- alpha (α): Speed limiting factor.
- minimization: Boolean indicating whether the optimization is a minimization or maximization.

The PSO Parameter Selection paper in the Manuals folder gives a summary of good PSO parameter values based on the number of parameters being optimized. Table 1 on page 7 gives a good summary of the results.

6 GIT

6.1 Rationale

Git is a distributed control system that allows for tracking and distribution of source code. It is currently used to track the entire PAT framework (link can be found here: `https://github.com/vbarua/QDG-Lab-Framework`). Now while the advantages of using git are numerous (read about them here `http://git-scm.com/about`) the main use for this project would be for future support services. If at any point in the future the framework stops working or you desire new features, an up to date git repository would allow an off site developer (ie. Victor) to quickly and easily get the current version of the framework and see exactly what changes have been made. Now while the command-line version of git can be daunting, the free Source Tree application (`http://www.sourcetreeapp.com/` makes it very easy to use. I would also recommend reading the first three chapters of this book `http://git-scm.com/book` to get a feel for the basics of git and its terminology.

6.2 Account Details

A user account has been setup on `https://github.com/` to allow the lab to contribute to the project. The username is *QDGLab* and the password is the same as that for QDG-GATEWAY machine.