Learning Goalie Robot

Semester Thesis SS 2007
Manuel Lang, Student Department of Computer Science ETH Zurich
malang@ethz.ch

Supervisor:
T. Delbrück, Inst. of Neuroinformatics, UNI-ETH Zurich
tobi@ini.phys.ethz.ch

This semester thesis is about a robot equipped with an asynchronous temporal contrast silicon retina. The objective of this robot is to behave like a miniature soccer goalie. It has to block incoming table tennis balls with his arm that is controlled by a standard servo motor. Furthermore, the goalie has to learn and adjust the parameter needed to control its arm and therefore adapt automatically to the given actual setup. This greatly simplifies the setup of the robot and reduces fiddling with various parameters.

I. Introduction

Traditional vision tasks often use frame based capturing video systems. These systems are limited by the given frame rate of the imaging devices. Consequently, high speed applications with short reaction time can only be implanted by increasing the frame rate what quickly results in an immense data volume that has to be processed to achieve the given vision task. Of course, this yields to increased component costs and decreases performance indices like power consumption.

Our approach for this goalie application is therefore to use an asynchronous event-based neuromorphic inspired silicon retina. [1], [2] This silicon retina differs from standard video capturing devices, by submitting local contrast changes asynchronously immediately when they occur instead of a frame based transmission. This is done by using a protocol called address event representation (AER) [3], which encodes the type of event (in our case increasing or decreasing contrast) and the position of occurrence (in our case the pixel position of our 128x128 pixel retina). This enables us to use standard PC equipment and still achieve fast reaction times.

Normal controlling tasks often use a closed feedback approach to cancel out typical and often complex setup intrinsic variables. This approach is highly practical for analog or hardware based controllers, but not for a software driven approach. In a software-driven approach this means to permanently read out the actual position and permanently change the controlling parameters to achieve the desired actual position(polling). Moreover a servo motor already implements a control loop. Hence, we tried to implement a better approach that learns the relationship between the servo motor input and the actual output directly. As a result, the software controller can directly apply the needed controlling values to achieve a given desired output, without having to observe and feedback the actual position of the actor (in our case the goalie arm). This simplifies the real-time positioning but comes with the cost of an additional learning task that has to be performed.

This work is going to show how to combine this AER based imaging approach with the learned controlling approach. Furthermore, we will describe the actual implementation and resulting changes to the JAER project [4] to implement this goalie application.

II. Setup

The silicon retina, which acts as the imaging device is mounted on the upper edge of a box that represents the goal. This means the camera is around 30cm above the table. On the bottom side of the goal box we mounted the servo motor so that the goalie arm that is parallel to the table is about 4 cm above the table.

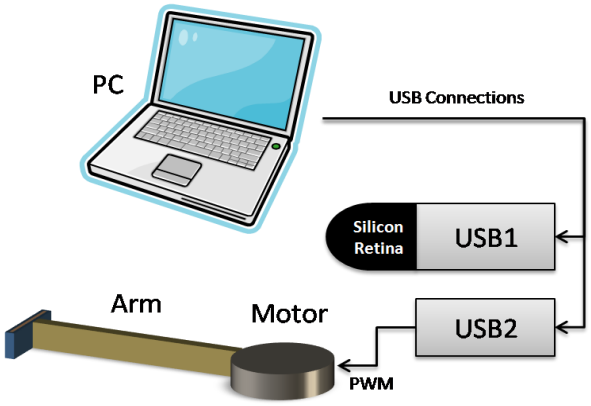

Figure 1 Picture of the goalie Setup

The length of the arm is about 30cm and a plastic hand is mounted at the end so that this hand is normal to the table and can be used to block the incoming balls. The box width, arm length and resting angle of the servo motor have to be adjusted in a way that the operational angle of the servo motor is large enough to cover the whole range of possible incoming balls. Small variation around the mid angle of the servo motor should result in an acceptable unconstrained movement in the visible area of the camera. However, it has not to be in the middle of the goal as this parameter is part of later learning.

The camera uses and 8mm wide range lens. For the self learning and calibrating functionality the camera has to capture the arm, hence we mounted the camera so that the arm is visible in the lower 30 pixels of the captured frame. The remaining part of the frame is used to track incoming balls. See *Figure 5.*

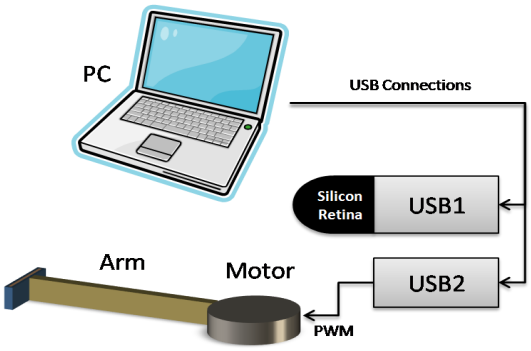

Figure 2 Schematic of the setup. USB connects a PC, Silicon Retina and servo motor

Both, the silicon retina and servo motor board are connected to a PC running the jAER software, which implements the actual tracking and controlling logic. The servo motor USB board converts commands over USB to a PWM signal that is sent to the servomotor.

III. ServoArm Class

Introduction

A Java class called ServoArm abstracts the movement and controlling of the servo arm. This class exposes a set of simple function that one can use to move the robot arm without having to deal with the insights needed to do so.

This class therefore implements the learning, which calibrates the servo arm. It can be triggered from outside to invoke a learning cycle.

It also allows logging the actual and desired arm positions to the Data Window that can than show this data as graph or a table.

Class design

Public Functions

`int getActualPosition()`
Returns the current position of the arm in pixels

`int getDesiredPosition()`
Returns the position the arm is going to move to in pixels. This value can be set by the setPosition function.

`void setPosition(int Pos)`
This function enables the servomotor when relaxed and starts a movement to the given position Pos. To do so the learned model is applied. When a learning cycle is in progress, it is aborted immediately to carry out the movement. This function returns instantly, in most cases before the given position is reached. If a subsequent call is done before the final position of a previous call is reached, the current movement is cancelled and the newly specified is carried out.

`void startLearning()`
Starts a learning cycle. During learning the servo arm is going to move and therefore cannot stay relaxed. Learning is done in a separate thread. Hence, this function returns instantly.

`void stopLearning()`
this function behaves like the function relax()

`void relax()`
This functions stop the movement of the arm immediately and disables the servo motor. This means, canceling of any scheduled movements and aborting of any running learning cycle.

`void startLogging()`
This function starts the logging functionality. After it is called, the actual and desired position is logged to Data View Window. Logging is done in a separate thread. Hence, this function returns instantly.

`void stopLogging()`
Stops the logging thread and deregisters data source in data Data View Window.

Super Class and Interfaces

`Class EventFilter2D`
ServoArm class has to register as filter, because it has process events during learning. See next chapter for details

`Interface Observer`
To initialize the filter.

`Interface FrameAnnotater`
Use to render useful information of filter state. It displays a bar for the desired and actual position of the arm. It also visualizes the arm tracking.

`Interface PnPNotifyInterface`
Interface is used to handle events that occur when dealing with the servo motor, like plug or unplugging the servo board.

Implementation Overview

The main part of the class is separated in two parts. The frontend, which exposes the public functions specified above, and a back end of private functions that handle the actual servo motor controlling.

The frontend functions are dealing with the logic needed to control the motor. They all operate on a higher abstraction level and use meaningful units like pixels to operate on. These functions are implementing the logic to enable the motors, start learning stop learning and so on. The positioning function setPosition uses the private helper function setPositionDirect as mid-layer. This function does not change the state (i.e does not stop learning), but does use the learned model to calculate the input value for actual hardware controlling function setServo. The learned model is encapsulated the function PositionToOutput.

All frontend functions check if servo motor hardware is properly working and connect before any other command is sent to the hardware. However, the relax function does not automatically reconnect to the hardware, because we assume that if we do not have an open connection to the servo board, we don't want to disable any motor.

The ServoArm class uses two additional threads that can run when needed. The logging and the learning thread. Both threads can be control by the corresponding start and stop functions. These functions more or less start and stop the threads. Both stop functions are blocking until the thread is actually stopped. Furthermore, the stopLearning function also disables(relaxes) the motor.

The logging thread simply registers two data sources to the Data Window; one ArrayList of data for the actual position and one for the desired position. After registering, the thread enters a loop which adds a new element at the end of the list. This is done in a specified interval, currently every 20ms. To save memory the logging data is cleared when 20000 elements are recorded. (This leads to a start over of the graph plot in the Data Viewer Window). Surely, all accesses to ArrayList have to be synchronized to avoid a race condition with the Data Viewer Windows, which reads out the same lists for painting the plots or displaying the data table.

Learning

Motor value and Pixel value relationship

The actual, learning model is surprisingly easy. We could achieve good results by simply using a linear function with two parameters. This means the relationship between the pixel position and motor input is given by the linear function

$$input_{motor} = k_{learned} * position_{pixel} + d_{learned}$$

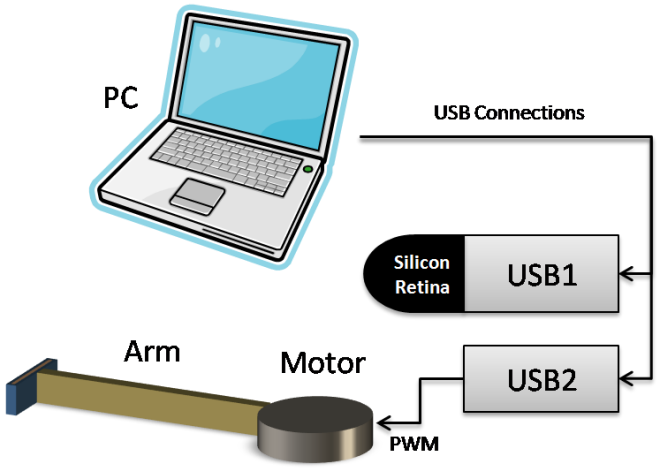

Figure 3 Shows the result of the learning. Each Point was sampled during learning. The line shows the linear relation between observed positions and motor input values.

Where $input_{motor}$ is a value between 0-1 that is sent to the motor, $position_{pixel}$ is the given desired position in pixels and $k_{learned}$, $d_{learned}$ are the parameter that are going to be learned. The $input_{motor}$ value describes the absolute angle that the servo motor should try to reach ($\alpha := input_{motor}$). Because, the desired position is the projection of the robot hand (end of the arm) to the x axe, mathematical justification would lead to a trigonometric relationship between this projection and motor angle, however this linear model proved to be accurate enough.

Sampling and Linear Regression

The learning of this linear model, therefore reduces to simple statistic problem namely fitting a regression line. First, we have to gather the samples we want to fit this model to.

This sampling is done by moving the robot directly to a known angle (send a value between 0-1) and then read out the observed position of the hand. We describe the tracking of the robot arm in more detail later, but for now assume we have access the observed position of the arm in pixel as it is seen by the silicon retina. Though, we limited our self to a smaller set of possible input angles. Instead of using the completely possible range from 0 to 1 we let the user specify the sampling range (`setLearningLeftSamplingBoundary` and `setLearningRightSamplingBoundary`). This is done to account for different setups and to avoid sampling outside the meaningful range for a given setup.

For every sample cycle, we can construct a Cartesian point (p, α) (where $p := position_{pixel}$). If we have a cloud of such samples, we can start to fit our learning model(linear function) to these observations.

This is done be utilizing the well-known statistic formulas [5]:

$$\mu_p = \sum_{i=0}^{n} p_i \qquad \mu_\alpha = \sum_{i=0}^{n} \alpha_i$$

$$\sigma_x = \sum_{(i=0)}^{n} (p_i - \mu_p)^2$$

$$\sigma_{xy} = \sum_{i=0}^{n} (p_i - \mu_p) * (\alpha_i - \mu_\alpha)$$

Then the two unknowns of the linear equation are:

$$k_{learned} := \frac{\sigma_{xy}}{\sigma_x}$$

$$d_{learned} := \mu_y - k_{learned} * u_x$$

To save memory and calculation time, we limited the number of stored samples to 100. When this number is reached the oldest sample is removed.

Implementation Details

As already mentioned, learning is implemented as an extra thread inside the ServoArm class. The thread is initiated when the function startLearning is called and stopped when stopLearning is called. The class implementing this thread is a private subclass of ServoArm. That sub class is instantiated only once when the ServoArm class is instantiated and reused for later invocations of the thread. The advantage of this approach is that this learning class can store and remember result across different learning cycles. Specifically, this is done to remember the samples of previous learning cycles.

A single learning cycle works as followed:

1. Check if learning is still necessary
   a. Move arm to random position by using high-level pixel based moveFunction
   b. wait for 2 seconds
   c. read out actual position of hand
   d. repeat a-c 5 times and calculate average error.
   e. stop learning if average error is good enough (smaller 5 pixels)
2. Select a random number inside sampling range
3. Send this value to servo motor (directly)
4. wait 1 second (time to move)
5. read out actual position of hand
6. add sample to set of samples (remove old sample)
7. if we have 8 new but at least 2o samples then do linear regression and go to 1
8. else loop back to 2

Learning is stopped whenever the learned model is accurate enough or an interrupting movement has to be performed. When learning is stopped, it does not start automatically again.

Learning Parameters

The only parameters for learning exposed to the users are the sampling range order. Normally these values are around 0.4 for the left and 0.6 for the right border.

There is no other parameter that has to be tuned by the user; however, there are few internal parameters. First, we us some fixed time values during learning whenever we have to wait. Secondly, we have a fixed accuracy value of five pixels whenever we check for the learning error and a fixed number of 5 runs for averaging this error. When collecting the samples, we have some additional internal parameters like, maximal number of samples(100) and number of new samples before a new model fitting is performed (8).

Tracking

ServoArm class is inherited from EventFilter2D. This allows the class to be put in the filter chain. This is needed to track the servo arm of the goalie.

We use the already implemented tracker RectangularClusterTracker. This tracker implements the normal filter interface. Therefore, we can forward any AER events we get from the retina to this standard tracker. However, first we use a XYTypeFilter to limit the input to the rectangular range were we expect the servo arm to be. Normally, this will be inside 30pixels range from the bottom and is specified by the user.

The tracker and the XYType filter are put into an enclosed filter chain of the ServoArm class. Therefore, we can easily forward events to both of them. It is important to realize, that the ServoArm filter itself is only an observing filter. Hence, any events passed to ServoArm should go through completely unchanged. Especially, the XYTypeFilter should not throw any events away, since they are needed for later processing (i.e. the goalie class)

Overshoot Protection

Introduction

Tests showed that the goalie arm tended to overshoot. That means, when the arm is moving to a specified position (or angle) it first moves a bit further, before the control loop inside the servo motor starts to correct for this error. This leads to a swinging behavior around the desired position.

Consequently, we had to improve our movement algorithm to account for this behavior. The idea is simple; we do not send the final desired motor angle directly to the servo motor. Instead, we divide each movement in two steps. First, we sent an angle that let the arm move 80% of the desired movement. 100ms after sending this intermediate step we send the final position to the servo motor. By doing this, the servomotor tries to reach the 80% position first and has already slowed down when the next command is sent. The desired final posi-

tion is then reached with slower velocity and as a result overshooting is reduced.

Implementation

Java provides a timer functionality. This allows to schedule tasks for later execution. We used this functionality to implement the overshoot protection. Whenever a new desired position has to be applied, we send the 80% intermediate step directly to hardware. Then we schedule a function that will send the final movement. The delay to this execution is set to 100ms. We also set the current desired position value, which is a private field of the ServoArm class, already to the complete 100% position value.

After 100ms the timer runs our scheduled function. This function simply sends the final position to the servomotor. However, we first have to check if the desired position is still the current one. It could have happened that in the meantime a new movement already started and therefore this function should not send the command down anymore. We can check for this case by comparing with the current desired value field that should still have the value stored that was saved when this function was scheduled.

Whenever the motor is going to be relaxed, we also have to clear the list of scheduled functions to avoid reactivation.

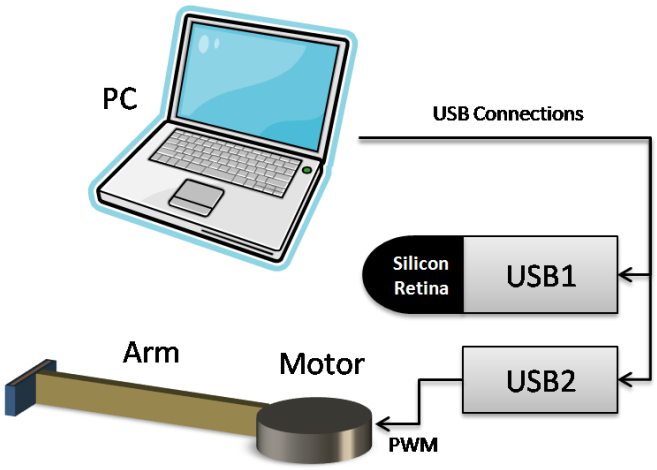

Figure 4 The overshoot behavior of the arm. Note the intermediate steps in the desired positions, which are resulting from the overshoot protection.

Future work

This servo arm implementation only supports one servo motor. Future work could be to control more than one motor, which means to allow more joints. The difficulty of the problem would then increase to an inverse kinematics problem [5].

A more dimensional movement would also allow more precise controlling of the incoming balls. This could lead to a system able to play more advanced games like real table tennis. [6]

IV. Goalie Class

Introduction

The goalie class implements the logic needed to behave like a goalie. Namely, tracking incoming balls, calculating the strike position and using the ServoArm class to move the goalie arm.

The goalie class was already implemented by a previous project. [4] Therefore, we could reutilize most parts. However, since the control of the servo motor was moved to the ServoArm class, we could clean up the obsolete code.

The goalie class also has to decide when to start a new ServoArm learning cycle and when to relax the goalie arm.

Implementation

The goalie class is inherited from EventFilter2D. This enables us to process any event sent by the retina. Again, it's important to realize that the goalie class is an observing filter and therefore should not change any incoming events.

The goalie call utilizes the XYTypeFitler, RectangularClusterTracker and ServoArm Class. The incoming stream of events are routed into two different sub chains. One sub filter chain only includes the ServoArm class. This is used for tracking the servoarm and therefore allowing goalie to control his arm.

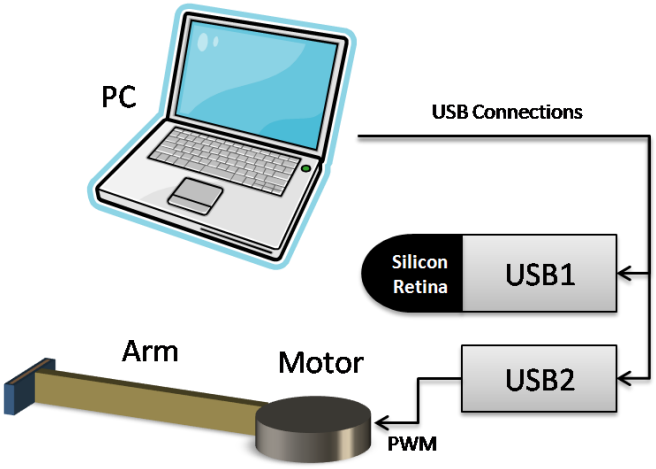

Figure 5 View from the silicon retina. Blue boxes show tracking area for arm (lower box) and balls (upper box). Tracked objects are annotated. The disired (white) and actual (blue) position of the arm is shown in the middle.

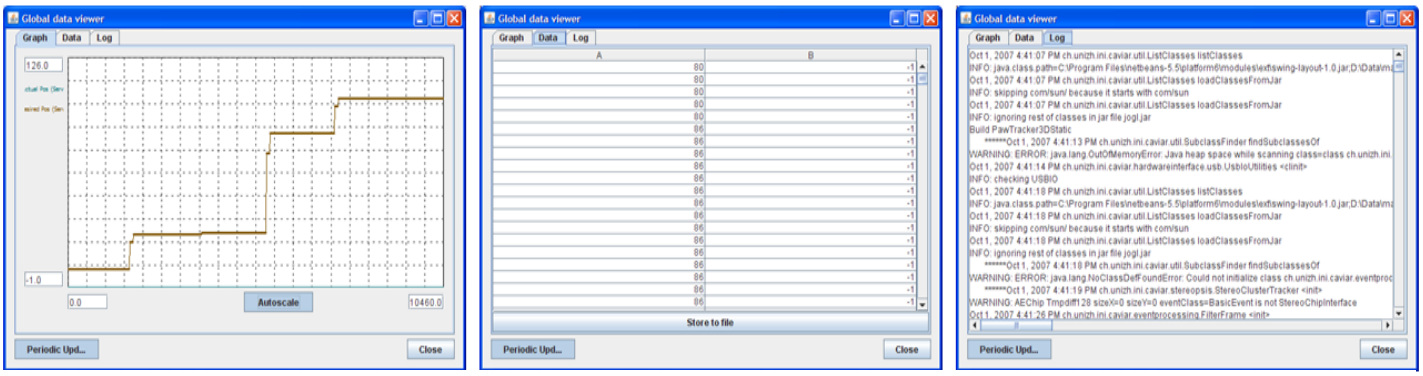

Figure 6: Shows the User Interface of the Data Viewer Window. Left: the graph tab which used to plot functions; Middle: Data tab which shows the data points from the plot; Right: Log Tab, shows the output of JAER

The second filter sub-chain is used to track incoming balls. Hence, the XYTypeFilter is used to crop away the servo arm. After that the remaining events are passed to the RectangularClusterTracker filter. This filter is parameterized to effectively track incomings balls. These settings can be obtained from the previous goalie implementation.

The goalie now calculates the expected point of impact of the the table tennis balls, which are tracked by the tracker. This is done by simply extrapolating the linear movement of the balls and intersect it with the y=30 line, which is the tangent line to the servo arm action scope. We also can calculate the expected impact time and therefore react first on the ball that will impact soonest.

Whenever a ball is tracked, a timer is reset. If no balls are tracked and the timer reaches a user predefined value a new learning cycle in the servo arm class is invoked by the goalie class. Therefore, learning starts when no playing is going on.

The timer is also used to relax the motors when no balls are in the play field after a short timer.

V. Data Viewer Window

Introduction

We added a new user interface window to jAER, which allows other part/classes of the program JAER to register data sources. This data source is then plotted or displayed in a table by this new data viewer window.

The data viewer window also redirects the normal system console output, which enables the user to view them inside the user interface instead of having to start the program from a console.

For that reason, the user interface displays three register tabs. One for displaying the registered data sources in a graphical chart. One for displaying the registered data sources in a table and one to display the console output. See *Figure 6.*

All plots/data are collected and displayed in one data viewer window. This gives the possibility to compare data from different viewer windows and or filters. However, this has to be considered when a filter is registering a data source, because it is possible that the same filter is already running in a different viewer and therefore a name conflict would occur if both use the same name to register the data source. Hence, it is recommend including the viewer name in the data source name when registering.

User Interface

Graph Tab

The graph tab allows plotting the registered data sources. On the left side all registered data sources are displayed as checkboxes. By selecting or deselecting a checkbox one can display or hide a given data source. The textbox at the end of the axis allows specifying the range on this axe. By selecting autoscale, this is done automatically and chosen so that all plots fit to the window.

Data Tab

The data tab is used to display data source in a table. This tab also allows to store this table to a coma separated file. This is especially useful if you want to import it later into different programs, for example to a spreadsheet program. The order of columns is the same as the order of checkboxes in plot window. However, XY data plots exist from two columns next to each other.

Log Tab

The log tab is used to print console output to a window. A user can therefore view them inside JAER itself, without having to switch to an operating system console. This is especially useful when the program is started form a graphical shell directly like Windows or X, where now text console is present.

Implementation

General

The data viewer functionality is implemented in the class JAERDataViewer. This class is composed from some subclasses; namely, GraphData, DataTable, GraphPanel and the enums Datatype, Linestyle. We will discuss all classes in more details I in following paragraphs.

JAERDataViewer

The class JAERDataViewer is inherited from JFrame, and therefore it is normal Java Dialog. This Dialog was designed using the GUI designer from NetBeans. We added 3 tabs as described above with the designer. This class implements all the logic used to handle the user interface, in particular the handling of mouse and keyboard events.

The main class JAERDataViewer uses GraphPanel to display the plots. GraphPanel class can be uses like a normal JPanel because it is implemented on top of the JPanel class. When Periodic Updated is selected by the user, a timer implemented in JAERDataViewer invokes a repaint of the GraphPanel and the data table every 0.5 seconds.

JAERDataViewer is responsible for registering new data sources. That means it exports a set of functions that other classes can use to register new plots.

The **addDataSet** functions are used to register new data sources. The most general function takes 7 parameters:

String Name: the name and key for later usage of this Datasource. This is used to label a plot in the user interface and has to be used again to delete a plot with removeDataSet.

ArrayList<Double> x: the list of X coordinates.

ArrayList<Double> y: the list of Y coordinates. This object is also used to synchronize access. y is locked whenever the data viewer is accessing either the x or the y list.

double samplingRate: This parameter is used in YScrolling mode to calculate the x axe labeling. It is ignored in XY or YScaling mode.

_DataType dataType: This parameter specifies the type of plot.
In XY mode the x and y list are used to plot the data. The data is interpreted as a set of (x,y) points. All points are connected by a straight line if style is Line or PointLine. In YScaling mode only the y list used. The first list entry is on the very left side of the plot. The last entry is plotted on the very right side of the graph panel. The remaining y values are distributed equally between the left and right side. In YScrolling mode only the last n y entries are printed. n is the width of the plot panel in pixels. Therefore, this mode behaves like a real-time chart.

LineStyle style, Color color: These two arguments are specifieng the type and color of the lines. Style is either Point, Line or PointLine.

JAERDataViewer also exposes simpler addDataSet function that can be used to simplify the usage. However, they all use the function described above as a backend.

A registered data set can be removed by calling **removeDataSet** and specifying the name that that was used when registering.

GraphPanel

GraphPanel is implanting a JPanel and extending it with the functionality to draw registered data sets. Therefore, the paint method is overwritten and changed so that the registered array lists are used to draw a chart.

This panel implements all functionality needed to draw the chart. This includes drawing the grid, the plots them self and the auto scale functionality. This panel can also inform observers when the axes are rescaled. This is used by JAERDataViewer to change the content of the axe text boxes when such an events happens.

Data Table

The data table is implemented with a normal JTable. However, to change the behavior as needed a new AbstractTableModel is implemented. This AbstractTableModel, called DataTable, provides the functionality to build the table from the registered data sources. This means it implements functions like getValueAt which returns the value in the table at specified location. DataTable also exports the function storeToFile that is used to save the table to a comma separated file.

Logging

The GUI of the logging tab is a normal JTextArea. The redirecting functionality of the err and out stream is implemented in the subclass StreamSupport which extends the java class OutputStream. This class implements a write function which then copies every character to the JTextArea. StreamSupport objects are then replacing the normal system.out and system.err objects this leads to a redirection of all console output to the JTextArea.

Usage

The data viewer windows can be accessed by any other class through the static member GlobalDataViewer of the JAERViewer class. Therefore, an example usage to register a real-time chart looks like:

```
JAERViewer.GlobalDataViewer.addDataSet("name",
actPos, interval, true);
```

This simple calling interface allows to extend many filters with a plot feature very easily. Next to registering as described

above, an implementation simply has to fill one or two ArrayList with data. Surely, this should be done in a synchronized manner (by locking the y list).

VI. Bibliography

1. *Time-derivative adaptive silicon photoreceptor array.* **Mead, Delbrück and Carve.** 1991.

2. **Lichtsteiner, P., Posch, C. and Delbrück, T.** A 128×128 120dB 30mW Asynchronous Vision Sensor that Responds to Relative Intensity Change. *IEEE ISSCC Digest of Technical Papers.* 2006, pp. 508-509.

3. **Lazzaro, John, et al.** Silicon Auditory Processors as Computer Peripherals. [book auth.] Stephen José Hanson and Jack D. Cowan and C. Lee Giles. *Advances in Neural Information Processing Systems.* s.l. : Morgan Kaufmann, San Mateo, CA, 1993.

4. **Lichtsteiner and Delbrück.** jAER. *SourceForge.* [Online] 2007. http://jaer.wiki.sourceforge.net/.

5. **Weisberg, S.** *Applied Linear Regression.* New York : Wiley, 1980.

6. **D'Souza, Aaron, Vijayakumar, Sethu and Schaal, Stefan.** Learning Inverse Kinematics. *Proceedings of the 2001 IEEE/RSJ.* s.l. : International Conference on Intelligent Robots and Systems, 2001.

7. **Matsushima, Michiya, et al.** A Learning Approach to Robotic Table Tennis. *TRANSACTIONS ON ROBOTICS, VOL. 21, NO. 4, AUGUST 2005.* s.l. : IEEE, 2005.

8. **Delbrück, T. and Lichtsteiner, P.** Fast sensory motor control based on event-based hybrid neuromorphic-procedural system. *ISCAS 2007.*

VII. Figures