

Solo_Predictor Reference Manual

From Eigenvector Documentation Wiki - Dec 1, 2009

Contents

- 1 Introduction
- 2 System Requirements
- 3 Features and Supported Methods
- 4 Interface Specifications
 - 4.1 Introduction to Socket Interfaces
 - 4.1.1 End-of-Message Indicator Option
 - 4.1.2 POST Protocol Option
 - 4.2 ActiveX and .NET Interfaces
 - 4.3 Wait-For-File Interface
 - 4.4 Single- and Multi-Client Servers
- 5 Installation and Configuration
 - 5.1 Installation
 - 5.2 Configuration
 - 5.2.1 Status Window and Controls Options
 - 5.2.2 Log File
 - 5.2.3 Server Connection Options
 - 5.2.4 Incoming Message Format and Timeout Settings
 - 5.2.5 Wait-For-File Options
 - 5.2.6 Output Format Options
- 6 Script Construction
- 7 Appendices

Introduction

Solo_Predictor, from Eigenvector Research, Inc. (EVRI) is a stand-alone model application engine which applies models created by PLS_Toolbox or Solo. Solo_Predictor features a simple and flexible scripting language, platform- and operating-system-independent interface, and an inherent distributed-computation design.

This documentation describes the setup and use of Solo_Predictor and explains the script language used to issue commands.

System Requirements

Solo_Predictor requires the following:

- Operating system:

Windows 2000, XP, 2003 server, or Vista

MAC OS X (Intel only)

Linux (Intel only)

- 200 MB Disk Space
- 100 MB RAM (recommended)

Features and Supported Methods

Solo_Predictor is a prediction engine which supports importing of data and models from an external source, application of those models to the data, and retrieval of the values from the prediction. It supports predictions for all methods which produce standard model structures in PLS_Toolbox and Solo. This includes all methods in the Analysis GUI (including, but not limited to, PCA, PARAFAC, MCR, Purity, PLS, PCR, MLR, PLSDA, SIMCA), Calibration Transfer GUI, and any other PLS_Toolbox command-line functions which produce standard model structures.

Solo_Predictor also supports:

- All preprocessing methods available in the custom Preprocessing GUI.
- Missing data replacement (where supported by the model type)
- Variable pre-alignment to model (handles resampling, extra variables, missing variables)
- Importing all data types supported by the Analysis GUI and Workspace Browser including, but not limited to:

Comma-, tab-, space-, and other delimited text files (.csv, .dat)

X,Y... delimited files (.xy)

Excel spreadsheets (.xls, .xlst)

Thermo-Galactic SPC files (single and multfile formats) (.spc)

Hamilton Sundstrand files (.asf, .aif)

Horiba JY files (various)

JCAMP (simple single-record formats) (.jcamp .jdx)

XML (Eigenvector XML data format) (.xml)

Matlab .mat files (.mat)

Note that Solo_Predictor does not support execution of custom, user-defined MATLAB® scripts or commands. Such functionality requires a full MATLAB license. Please contact Eigenvector Research for more information on using Solo_Predictor in a MATLAB environment.

Solo_Predictor can be connected through a socket interface using TCP/IP, through an ActiveX or .NET object, or operate in a wait-for-file mode. It can send results to a client and/or write to an output file. Solo_Predictor also maintains a text-based log file to aid with diagnosis of problems.

Interface Specifications

In this description of the Solo_Predictor interface, the term "client" refers to a user-specified application which is requesting a prediction and the term "server" refers to Solo_Predictor. The client is often a distributed control system (DCS) or other data collection software (instrumentation software, etc) but can be any application which needs to apply a multivariate model to data. In general, the client issues one or more commands to Solo_Predictor either by passing data or by describing where data can be retrieved from. Additional commands are passed to instruct Solo_Predictor how to process that data and what results should be returned. See the Scripting Language section for details on the scripting language used for the instructions.

Introduction to Socket Interfaces

Solo_Predictor operates using standard TCP/IP (Transmission Control Protocol/Internet Protocol) communications over "socket" connections. Sockets are available on all operating system platforms (Windows, Mac, Linux) and are the same technology used in most Intranet and Internet communications including http, ftp, and other familiar inter-computer systems. They are also used for some "plug and play" hardware devices. Simply put, sockets are a general method to pass messages between two programs.

Although socket connections are most often used *between* computers, they can also be used when the client and server reside on the *same* computer (and even when that computer is not networked). When connecting two programs on the same computer, sockets are similar to other familiar inter-program communication systems (e.g. DDE or Active-X) with these added advantages:

1. Sockets are completely platform independent. The same communication methods are used on all operating systems and hardware. They can also be used across mixed operating systems and platforms (e.g. Windows to Linux.)
2. Most modern languages have some sort of provision for socket communication and require no proprietary technology to implement.
3. Socket technology allows the client and server to be located on the same computer or separate computers connected by a network. The identical software and setup are used in both cases. The only modification needed is to provide a remote IP address or name for the server. As a result, sockets also inherently allow for distributed computation.

The procedure of communication over sockets is well described in many places. The basic procedure is:

1. The client opens a socket connection between the client and server. This requires knowing the IP address of the server's computer (use "loopback" or "127.0.0.1" if the server and client are on the same computer) and the port number on which the server is "listening."
2. The client sends a command to the server. The end of the message is indicated when no additional characters are available.
3. The server receives the command and performs some operation.
4. The server returns a response to the client often containing either a simple acknowledgement of the message or possibly some additional data or results.
5. The socket connection is closed.

The messages passed to Solo_Predictor are passed in plain text, but the ability to pass XML to describe some more complicated data types also exists. The response from Solo_Predictor can be in any of a number of formats including plain text, XML, or HTML. In addition, Solo_Predictor also permits some standard HTTP-format (i.e. web browser-style) input and output messages. For more information on the message format, see the "Scripting Language" section in this manual.

See Appendix C Solo Predictor Example Connection Code for socket-connection coding examples.

End-of-Message Indicator Option

In some cases, a system has a high load (many programs running) or the messages being transferred are large. In these cases, the message transferred by the client may be broken up into smaller pieces. This may cause Solo_Predictor to believe the message is complete before it has received the entire message. In these cases, Solo_Predictor can be told to expect an end-of-message (EOM) character or string (e.g. "[EOM]") and it will wait to process a message until it sees that string arrive. See Incoming Message Format and Timeout Settings for how to set an EOM string.

POST Protocol Option

Solo_Predictor also accepts the common HTTP POST protocol for incoming messages. This format specifies the expected length of the message and, thus, allows messages to be split into segments because Solo_Predictor will not process the entire message until the received message is that length. See this external page for a simple example of the POST protocol format (<http://www.jmarshall.com/easy/http/#postmethod>) . Although standard POST format allows specification of different content types, the only Content-Type header which Solo_Predictor currently supports is text/plain. The following gives an example of a valid POST message for Solo_Predictor:

```
POST . HTTP/1.0
Content-Length: 15
Content-Type: text/plain

data='[1 2 3]';
```

Also note that outgoing messages from Solo_Predictor are never "chunked" (split into several pieces) nor do they ever use the POST format.

ActiveX and .NET Interfaces

For client applications which cannot or do not want to use sockets, Solo_Predictor provides both an ActiveX and .NET suite of objects called EigenvectorTools which can communicate with Solo_Predictor without the client having to implement socket interface code. EigenvectorTools must be installed on the same computer as the client application, but Solo_Predictor can still be located on the same computer or on a separate computer (if the socket option is used). Please note that EigenvectorTools are only available on Windows. Other platforms must use Sockets to communicate with Solo_Predictor.

For information on using EigenvectorTools, see the help page EigenvectorTools. Note that although the EigenvectorTools page makes reference to accessing graphical user interfaces (GUIs), Solo_Predictor does not allow access to the GUIs. Only the creation of data objects and application of models.

Wait-For-File Interface

Solo_Predictor also offers a basic wait-for-file method of interface. This feature is designed for compatibility with legacy systems which may not offer flexible interfacing and allows a client to trigger an analysis by simply dropping a readable file into a specified folder. Solo_Predictor can be configured to write a response file for the client to read the results of the analysis. For more information on this option, see the Installation and Configuration section and the Script Construction section.

Single- and Multi-Client Servers

Upon starting up, Solo_Predictor will automatically identify itself ("imprint") with the first client computer that makes contact with it. After imprinting, only that computer will be able to send commands to the server. This is true if the client and server are on the same computer, or on separate computers. Solo_Predictor can only be reset to respond to another client by restarting the server.

Some licenses will permit more than one client computer to access the predictor simultaneously. Thus, a single predictor can be installed on a centrally-located, networked computer and serve a number of clients on different computers (or multiple clients on the same local computer). Note that although multiple clients can make connections and request predictions, the following conditions are put into place:

1. Each client normally has its own workspace to store data and results. That is, one client cannot normally access the workspace of other clients. This can be disabled if, for example, multiple clients are contributing to the data used to make a prediction or when a remote client will be used to interrogate the workspace of another client. See the Installation and Configuration section for more information on workspace options.
2. In order to assure the fastest response for a given client, Solo_Predictor will only execute one client's request at a time.

Please contact Eigenvector Research, Inc. for more information on multi-client licenses.

Installation and Configuration

The following section describes the options available for configuring Solo_Predictor.

Installation

Solo_Predictor is packaged in several different ways depending on the platform on which it is being installed. Follow the instructions provided with the downloaded software to install on the appropriate platform.

Solo_Predictor is typically run by a start-up process so that it is always available, however, it can also be started "on-demand" by simply executing the Solo_Predictor file or shortcut (again depending on the operating system). The options for stopping or restarting the server depend on the configuration of the Status Window (see below).

The server's IP address depends on the local network setup. If the client is running on the same system as Solo_Predictor, then the loopback address (127.0.0.1) can be used for both client and server. The port number is configured as described below.

If, however, Solo_Predictor is on a different computer than the client, the client must make a connection into the computer running Solo_Predictor. Normally this is done by IP address but most sockets provide some means for looking up an IP address based on the computer name. If dynamic IP addresses are being used, it is recommended that the Solo_Predictor computer be set up with a preference for a given IP address. However, if the IP address does get changed, the client will need to be pointed to the new address.

For more information on programming socket connections, see Appendix C:
Solo_Predictor_Example_Connection_Code.

Configuration

All configuration of Solo_Predictor is accomplished through the defaults.xml file which is located in the program's main folder. This XML file contains a number of tags which can be edited by the user. Note that changes in this file will not be read by Solo_Predictor until the server is stopped and restarted.

The tags within the <socketserver> tag control the server settings. In each case, an options value is provided using standard XML notation:

```
<optionname>value</optionname>
```

In addition, inside each opening tag, several attributes are set:

```
<optionname class="numeric" size="[1,1]">1</optionname>
```

The "class" attribute should not be changed from the given value. The "size" attribute is informational only and can be omitted.

The following are the user-modifiable options. The expected class attribute is included in parentheses.

Status Window and Controls Options

These options control functionality of the Solo_Predictor status window.

- **controls** (class="string"): Manages the display and functionality of the status window. Valid settings include:
 - **none**: no status window will be given and all controls are hidden.
 - **status**: status window is shown, but all server controls are disabled.
 - **limited**: status window is shown and only the "restart" control is enabled.
 - **full**: status window is shown and all controls (stop/start/restart/exit) are enabled.

Except when "full" settings are used, the only means to stop and/or restart the server is by using operating-system-specific process kill commands ("Program Manager" in windows, the "Activity Monitor" in OS X, and the kill command in linux or unix). Default is "status".

- **max_screen_lines** (class="numeric"): Defines the total number of past message lines displayed on the (on-screen) status window. Default is 20 lines.
- **pulseperiod** (class="numeric"): Defines the number of seconds between "pulse" messages in the status window. Default is 15 seconds.

Log File

These options control the log file and the level of detail and age of messages retained.

- **log_severity** (class="numeric"): Defines the minimum message "severity" which will be reported in the log file (on disk). The level must be one of the following:

0 = log all messages

1 = log all startup, shutdown, rejected connection and fatal error messages

2 = log fatal error messages only

3 = log no messages (disable logging).

The default level is 1 (one).

- **max_log_size** (class="numeric"): Defines the maximum log file size (in bytes). Solo_Predictor will discard old messages to keep the log file from exceeding this size. Default is 50000 (50 Kb).
- **logfile** (class="string"): Gives the path and filename to use for the log file. By default, this is solo_pred.log in the user's temporary directory. The exact location of the temporary folder depends on the operating system. For example, this is usually:

Windows XP: \Documents and Settings\username\Local Settings\Temp.

Windows Vista: \Users\username\AppData\Local\Temp

Server Connection Options

These options control the behavior of the socket server and the kind of connections it will accept.

- **port** (class="numeric"): Defines the computer port on which the socket server will respond to requests. This value should be changed with great care as some sockets are used by the operating system and other software. The default port value of 2211 is selected to minimize conflict between known port uses. Additional ports which might be of use include: 2210, 2212, and 2005. Contact Eigenvector Research for more information on valid ports.
- **loopbackonly** (class="numeric"): If set to 1 (one), the server will only respond to a client which is located on the same computer as the server. All external requests will be ignored. A value of 0 (zero) will respond to any IP address (see also validip option). Default is 1 (one).
- **validip** (class="cell"): Gives a list of valid IP addresses to which the server may respond. If empty, any IP address client is permitted to contact the server (unless the loopbackonly option is set to 1 (one)). Remember that the server is limited to a given number of clients (usually 1 (one)) and once it has been contacted by that many clients, it cannot respond to any other clients. This setting only limits the clients who can contact the server before it has imprinted on a given client.

The ip addresses must be supplied as separate items each inside a set of <td></td> tags with all <td> tags enclosed in a set of <tr> tags. For example:

```
<validip class="cell">
  <tr>
    <td>10.0.0.1</td>
    <td>10.0.0.2</td>
  </tr>
</validip>
```

- **privateworkspace** (class="numeric"): If set to 1 (one), each client will have its own workspace to store objects and no client can access another client's objects. If set to 0 (zero), each client accesses the same workspace. A client may access and/or overwrite other client's objects. This may lead to unexpected results (if a given client expects a model to stay loaded but other clients are using the same object name and overwrite the model, for example). Default is one.

Incoming Message Format and Timeout Settings

- **eomstring** (class="string") End Of Message character or string. If non-empty, this character or string must be passed to indicate end of message. The same string will be appended onto any messages returned by the server. The use of an EOM string allows Solo_Predictor to function on higher-load systems or with large messages where the entire contents of the message may not be queued and delivered all at once. See Introduction to Socket Interfaces for more information. It is best to set a string which is very unique and will never show up in a common message, for example: ****EndOfMessage****
- **tickletimeout** (class="numeric") Number of seconds of delay allowed between opening socket and getting first character. At timeout, before sending client a space character. Required to tickle some clients into responding.
- **emptytimeout** (class="numeric") Number of seconds of delay allowed before receiving first message from client. At timeout, throws an empty packet message.
- **eomtimeout** (class="numeric") Number of seconds after which no more characters received indicates an end-of-message (generally for use with POST messages and EOMSTRING messages only).

Wait-For-File Options

Wait for file options control the optional Solo_Predictor wait-for-file engine. This engine will watch a given folder for a new file (with an optional specific file type). When a new file appears, the file will be automatically loaded as the object "data" and a specific script (stored in a disk file) will be executed. This script can use a :writefile command (see Script Construction section) to store results of an analysis in an output file.

- **waitforfile** (class="string"): either "on" or "off" (the default). When "on", the wait-for-file functionality is enabled (although the waitfolder and waitscript must also be non-empty strings for wait-for-file to operate).
- **waitfolder** (class="string"): defines the folder (local or networked) in which Solo_Predictor should look for new files.
- **waitfilespec** (class="string"): defines the file specifications (if any) to which the wait-for-file should be limited. For example, waitfilespec = "*.dat" will only recognize .dat files appearing in the wait folder.
- **waitscript** (class="string"): defines the filename containing the script to execute when a new file is found. This option must contain the entire path to the file. Note that the indicated script should expect to find the loaded data in the object named "data" in the current workspace.

Output Format Options

- **default_format** (class="string"): Defines the default response format. This is the output format used by the server if no format type is included in the request script. Valid types are: "xml", "plain" or "html". See Scripting Language for more information on these formats. Default is "xml".
- **writefilefolder** (class="string"): Defines the top-level folder to which writefile is allowed to write. Writefile command can ONLY write to this folder and any sub-folders of it. Empty string for writefilefolder = writefile is NOT permitted at all.

Script Construction

Solo_Predictor provides a simple, flexible scripting language with which clients can send instructions to load data, apply a model to that data ("make a prediction"), and retrieve results. For details, see the page: Solo_Predictor Script Construction

Appendices

The following additional information is available about using Solo_Predictor:

- Appendix A: DataSet XML Format - describes XML format used to create DataSet objects.
- Appendix B: Solo_Predictor Script Commands Summary
- Appendix C: Solo_Predictor Example Connection Code|

Retrieved from "http://wiki.eigenvector.com/index.php?title=Solo_Predictor_Reference_Manual"

- This page was last modified on 16 November 2009, at 23:15.

Script Construction

From Eigenvector Documentation Wiki

Contents

- 1 Workspace Objects
- 2 Script Commands
- 3 Importing Commands
 - 3.1 Importing From a File
 - 3.2 Creating Data From Script Text
 - 3.3 Creating Objects from XML
- 4 Application of Models and Preprocessing Commands
- 5 Requesting Return Values
 - 5.1 Retrieving Multiple Values
 - 5.2 Common Return Properties
- 6 Response Message Format
 - 6.1 :xml
 - 6.2 :plain
 - 6.3 :html
- 7 Write To File Command
- 8 Other Commands
- 9 Scripting Examples
 - 9.1 Reading data from file (local or network)
 - 9.2 Reading data from passed CSV
 - 9.3 Reading data from passed XML
 - 9.4 Wait for file and output to results file

Solo_Predictor provides a simple, flexible scripting language with which clients can send instructions to load data, apply a model to that data ("make a prediction"), and retrieve results. A typical exchange follows this sequence:

1. load data
2. load model
3. apply model to data (make a prediction)
4. return prediction results

This section describes the details of how to format a script, what commands are available and how the commands are used. The next section gives several quick-start example scripts which can be used as templates to perform some standard analyses.

Some familiarity with multivariate analyses and modeling are presumed by this chapter. The user is directed to the PLS_Toolbox and Solo Manual and Tutorial to learn more about specific modeling methods and multivariate analysis in general.

Workspace Objects

Each command in a script creates or operates on objects stored by Solo_Predictor for the client. Objects include:

DataSet Objects – contain data to be used in predictions.

Preprocessing Objects – contain instructions for how to apply preprocessing to a DataSet.

Calibration Model Objects – contain details on a calibration model from PLS_Toolbox or Solo.

Calibration Transfer Model Objects – contain details on a calibration transfer model from PLS_Toolbox or Solo.

Prediction Objects – contain results from applying a calibration.

When created, each object must be given a unique name (up to 64 characters in length) using only letters, numbers, and the underscore character (_). Object names may not contain spaces and may not start with a number ("a1" is allowed, "1" is not), but are otherwise unlimited. Giving a new object the same name as a previously existing object overwrites the original object.

All objects exist in a persistent "workspace" – it remains intact from one request call to another. In addition, this workspace is usually unique to the individual client so that no two clients can access the objects in another's workspace (however, see the privateworkspace option in the Installation and Configuration section.)

Script Commands

Script commands fall into these categories:

1. Importing commands – bring data or objects into Solo_Predictor

```
obj_data = 'content'
```

2. Model and Preprocessing Application commands – apply an object to data

```
obj_result = obj_data | obj_model
```

3. Return Value Request commands – request the value contained in an object.

```
obj_result
```

4. Response Message Format commands – set the output format for returned results and errors

```
:format
```

5. Write To File command – creates an output file containing results

```
:writefile
```

6. Other commands

```
:command
```

7. Comments – not processed by Solo_Predictor.

```
//comment  
#comment  
%comment
```

Each script can contain one or more commands, so more than one operation can be performed in a single call. Multiple commands are separated by semicolons. For example:

```
command; command; command
```

White-space characters in a command (i.e. spaces, tabs, line-feeds) are generally ignored. They can be included for readability but are not generally required. The exceptions to this rule are noted below (for example, in some of the data importing formats where white-space may be required.)

All script commands are also summarized Solo_Predictor Script Commands Summary.

Importing Commands

Nearly all scripts start with one or more importing commands. To perform a prediction, a client must request the import of data and a model (at a minimum). Importing commands create data, models, or preprocessing objects in the client's workspace. These can be created from:

1. Reading the object from one of a number of supported file types, or
2. Creating the object directly from text in the script.

In either case, the format of the command is the new object's name, followed by an equal sign and the content for the object in single quotes:

```
obj_1 = 'content'
```

The content takes various forms depending on the source of the object. Note that if the content needs to include a single quote, it should be escaped by adding a backslash in front of the quote: \' This is most often necessary when importing an XML object (see below)

Importing From a File

Any of the objects used by Solo_Predictor can be loaded from a disk file. This is the most common form of import command when a client program can only save data to disk, and is also the most common command used to load a model. A file that is stored either locally on the server's computer or on a network-mounted drive can be loaded by providing the path and filename (including extension) of the file enclosed in single quotes.

```
obj_1 = 'C://full/path/filename.ext'
```

Note that the path to the file is always relative to the server, not the client. Solo_Predictor will automatically recognize the filetype and use the appropriate file reader to import the file.

If reading from a MATLAB .mat file containing more than one variable, a specific variable to load must be

specified in the import command. This is done by appending a question mark followed by the variable name, all inside the single quotes.

```
'obj_1 = 'C:/full/path/filename.mat?variable'
```

This is only necessary when the .mat file contains more than one variable. If only one variable is present, that variable will be loaded without a specific variable name being specified.

The Features and Supported Methods section discusses valid file types, and other types may be available. In general, the list of file types supported by the Analysis GUI and Workspace Browser in Solo and PLS_Toolbox are also supported by Solo_Predictor.

Creating Data From Script Text

Data can be created from a list of comma-separated values by simply passing those values as text between the quotation marks:

```
'obj_1 = '1,2,3,4'
```

This would create an object that contains the values 1 through 4. Note that this form of input provides no means to assign labels or axis values corresponding to these values. As a result, no variable alignment or correction for missing or extra variables can be performed. It is assumed that the number of variables passed is appropriate for use with the model of interest.

Creating Objects from XML

Any of the objects used by Solo_Predictor can be created from an XML format that is supported by both PLS_Toolbox and Solo. In this form, the content of the import command is an object's XML description.

```
'obj_1 = '<obj>(XML formatted content)</obj>'
```

Although XML input is more complex, it enables the widest range of features supported by Solo_Predictor, including variable replacement and alignment. To simplify its use, a template XML file showing the tags necessary to create a DataSet object is discussed in DataSet XML Format. This template can be used to create and pass appropriate XML for these objects.

Any object that has been exported from Solo or PLS_Toolbox as XML (including DataSet, Model, and Preprocessing objects) can be stored and passed into Solo_Predictor to re-create the given object.

Application of Models and Preprocessing Commands

Once a calibration model, calibration transfer model, or preprocessing object has been imported into Solo_Predictor, it can be applied to a DataSet object using the application command. This command consists of an output object name, an equal sign, a DataSet object's name, the bar character, and the model or preprocessing object to apply. The output of the application command depends on the type of object being applied.

```
modifieddata = data_obj | preprocessing_obj  
modifieddata = data_obj | caltransfer_obj  
prediction_obj = data_obj | model_obj
```

When applying a preprocessing object to a DataSet object, the result is always another DataSet which contains the modified data. This DataSet can then be used in subsequent application commands or retrieved using a Return Value Request command, described below.

When applying a calibration transfer model to data, the result is a modified version of the original DataSet object that can be used in a subsequent command (much like with preprocessing objects.)

A calibration model applied to data outputs a prediction object. Prediction objects are similar in content to a model, but contain results specific to the application results. It is most often the fields of this prediction object which a client will want to retrieve as the final result of a model application.

Requesting Return Values

The eventual goal of most scripts is to return one or more values to the client. A large number of different statistics, results, and information are available. These are all stored as properties of the objects in the client's workspace. The specific value or values an individual client will need to retrieve depends largely on the model being applied and the intended use of the model. Some typical requests will be discussed later.

A script can specify the values to return by sending a return request command. This command specifies an object's name followed by a period and the name of the property (also known as "field") to return.

```
Obj.property
```

The specified object's property will be returned to the client when the script finishes. Properties may contain strings, single numeric values, numeric vectors, numeric arrays or complex objects. The format of the contents will be based on the Response Message Format described later in this section.

Note that the returned value will be the value of the object and property at the point in the script where the statement occurs. Changing or clearing the object or property *after* the request statement will not affect the returned value.

Retrieving Multiple Values

A given script can only return a single set of values. This can be from a single return request command or multiple "compatible" return request commands. Commands are compatible if the specified values can be concatenated horizontally into a row vector or matrix. To retrieve multiple compatible values, the script can simply include several return request commands within the body of the script (with terminating semicolons as is always necessary between commands.) For example:

```
'Obj_A.property1;  
'Obj_A.property2;  
'Obj_B.property1;
```

If the commands refer to objects or data tables which cannot be combined in this manner (e.g. different number of rows or incompatible objects), Solo_Predictor will return a script error.

If multiple incompatible properties need to be retrieved by a script, multiple connections will need to be made to Solo_Predictor retrieving the desired properties one at a time. Remember that because objects are persistent from call to call, the objects created by one call to Solo_Predictor remain available for subsequent calls. See the Scripting Examples for an example of retrieving multiple values. Another option to retrieve multiple outputs is the Write to File command discussed later in this section.

Common Return Properties

Model and prediction objects can be queried for a description of the properties which are typically used in predictions from the given model type. Sending the command:

```
Model_Obj.help.predictions
```

will return an XML description of properties of Model_Obj which the client program might want to make available to the user. This description is comprised of a single <tr> tag enclosing multiple <td> tags, each containing the description of an available property in the model including the following tags:

- **<label>** contains a text description of the property's contents
- **<field>** contains the full property name which should be used to access the given value. Add the contents of this field onto the string name of the prediction or model object followed by a period: pred.field
- **<dimension>** contains a string describing the type of return value provided in this property for the prediction of a single sample. This will be "scalar" (single value), "vector", or "matrix" and can be used by the client to ignore return value types which it cannot manage. "vector" means a single row of numbers. "matrix" means a table of values.

For example, the following XML fragment describes two fields available from a PCA model:

```
<tr>
  <td>
    <label class="string">Scores</label>
    <field class="string">loads{1}</field>
    <dimension class="string">vector</dimension>
  </td>
  <td>
    <label class="string">Hotelling's T^2</label>
    <field class="string">tsqs{1}</field>
    <dimension class="string">scalar</dimension>
  </td>
</tr>
```

The first <td> tag describes the Model_Obj.loads{ 1 } property which contains Scores and will be returned as a vector. The second <td> tag describes the Model_Obj.tsqs{ 1 } property which contains the Hotelling's T² (T-squared) value which will be returned as a scalar (single value).

The client can use this list to populate a table (or other GUI) with outputs available from the predictions using a given model. Note that the help.predictions list is the same from a model object and any prediction objects made from it.

Because only model and prediction objects provide self-description of useful properties, the following the following tables, organized by object type, are provided to guide the user to some of the standard properties which clients may be interested in retrieving. Note that some properties may contain additional indexing information in either "curly" braces { } or in standard parenthesis () and that this indexing must be included as shown.

Table 1. Commonly used properties for DataSet objects

Property	Description / Content
.data	The numerical data
.label{2}	Labels for the variables (if any)
.axisscale{2}	Numerical axis scale for the variables (if any)

Table 2. Commonly used properties for Prediction and Model objects

Property	Description / Content
.scores	The numeric scores of a model
.t2	The Hotelling's T-squared value. These values will generally be "reduced" so that a value of 1 is at the pre-defined confidence limit. See .detail.options.confidencelimit property to retrieve the confidence limit used for the reduced value.
.q	The sum-squared residuals (a.k.a. SPE) These values will generally be "reduced" so that a value of 1 is at the pre-defined confidence limit.
.prediction	The y prediction(s) for regression models.
.tcon	The Hotelling's T-squared contributions.
.qcon	The Q contributions (X-block residuals).

Response Message Format

All results returned by Solo_Predictor are text-based but the exact format of the text can be selected to best match the client program's ability to parse text. The format is selected using a format command which consists of a colon followed by a format keyword. A format command can be located anywhere within your script, before or after a return request command. As always, each command must be separated from other commands by semicolons. The following are valid output formats:

:xml

Selects the XML format. This consists of three tags: result, error, date.

result : Contains any output produced by a "return value" command. The class of the returned value is given in the class attribute of this tag. Standard classes are "string" and "numeric" but other complex objects may be returned. Numeric values are given in comma- and semicolon-delimited format where commas delimit row-wise elements in vectors and arrays and semicolons delimit column-wise elements in vectors and arrays. For example, an array with two rows of 5 numbers would be returned as:

```
-----
1,2,3,4,5;6,7,8,9,10
-----
```

In addition, the result tag will have a size attribute included with any string or numeric value. It will give the expected result's size in rows and columns. This can be used by the client to prepare an appropriately-sized

matrix for the parsed result and for error checking on the parsed result.

error : Contains a text description of any errors which occurred during the script execution. If no errors occurred, the error tag will be empty.

date : Contains the date and time the request was received by the server.

In XML format, a response with no errors and no results will return as an XML structure with result and error tags empty. Date will contain the relevant date information as usual.

Example:

```
<response>
<result class="numeric" size="[2,5]">
1,2,3,4,5;6,7,8,9,10
</result>
<error class="string"/>
<date class="string">Thu 06 Sep 2007 14:21:08</date>
</response>
```

:plain

Selects the plain-text format. If any errors occurred, this will be an error message starting with the text: "ERROR:" and will supersede any results.

When no errors occurred, the content of any return values will be given in a space- and linefeed-delimited format where spaces delimit row-wise elements in vectors and arrays and linefeeds delimit column-wise elements in vectors or arrays. For example, an array with two rows of 5 numbers would appear as:

```
1 2 3 4 5
6 7 8 9 10
```

Note that multiple spaces will almost always be used between row-wise elements.

In plain-text format, a response with no errors and no results will return as an empty (null) message.

:html

Selects an HTML-friendly format. This output is appropriate for display by any standard HTML-parser such as a web browser; All text is enclosed in preformatted text tags (<pre>). If any errors occurred, they will be displayed much like the plain format. Otherwise, the result will be included in an XML format (same as the result field in the xml format.)

If more than one format command is included in a script, only the last-encountered format command will be used.

The default format for the response is set by the default_format option. See the Installation and Configuration section for more information.

Write To File Command

Another method of returning results to a client is the write-to-file command. This command provides a simple way to create a specifically-named output file based on a user-created "template" file. This command is particularly useful when the client is unable to parse returned strings, when passing particularly complicated return values, or when Solo_Predictor is being used by legacy systems which expect a file response. See the example scripts section for an example of their use.

NOTE: By default, the `writefilefolder` option in the configuration file, `default.xml`, is blank. This disables the `writefile` command. This option must be configured before using the `writefile` command in any script.

The general format of the command is:

```
:writefile 'path/output.ext' 'path/template.tem' -append
```

When encountered in a script, this command specifies the output filename and extension ('path/output.ext', which should include the entire path to the output file), the name of the template file ('path/template.tem' also including a full path) and a flag indicating if the results should be appended to the end of the output file rather than replacing the file ("-append"). All values except the output filename are optional. If the -append flag is omitted, any existing output file is overwritten. If the template filename is omitted, a template file named 'output.tem' is expected in the same folder as the output file. Note that the only acceptable template file type is ".tem".

The template file can consist of static text and replacement keys. Static text is written to the output file exactly as it appears in the template file (including linefeeds, spaces, tabs and all other text characters). Replacement keys consist of any return request command, described earlier, inside square brackets []. When the output file is written, replacement keys are replaced with the value(s) indicated by the contained return request command. For example, consider the following template text:

```
1, "residuals", [pred.q]
2, "tsquared", [pred.t2]
3, "y-prediction [[conc]]", [pred.prediction]
```

This template would create an output file like:

```
1, "residuals", 0.3225352
2, "tsquared", 0.1212351
3, "y-prediction [conc]", 12.25252
```

Note the following:

- The `<writefilefolder>` option in the `default.xml` file configures the folder into which files can be written and template files may be read. If this option is set to empty (blank string), then no `:writefile` commands will be permitted.
- An optional formatting instruction can be included inside the replacement key brackets. This instruction consists of a simple `fprintf` instruction (`%_._f`) followed by a comma and the usual return request command. For example:

```
[%i,pred.prediction]
```

The percent sign can be followed by optional precision indicators `._` but must be followed by a format character (usually `i`, `f`, `g`, or `e`). For more information on formatting commands see: `SPRINTF`

Documentation (<http://www.mathworks.com/access/helpdesk/help/techdoc/ref/sprintf.html>|Mathworks)]
)

- When the return request command inside of a replacement key consists of more than one value, all values will be returned as comma separated list. To return in any other format, the template file must be hard-coded to access each value individually: [pred.prediction(1)] [pred.prediction(2)]
- To include square brackets as static text in a template, use double brackets [[and]].
- When using the -append mode, make certain to include any linefeed characters required. Without linefeed characters, each subsequent output will be appended to the same line.

Other Commands

Several other commands exist for various operations. Each consists of a colon followed by the command name and can generally be used anywhere within a script.

:include 'path/script.txt'

Include the contents of the specified file in the current script. The indicated text file (which can have any file name and extension but must include the full path to the file) will be executed by Solo_Predictor as if it had been passed explicitly in the current script.

This command permits a client to pass a simple :include command to invoke a possibly complicated script. The included script is always executed in the same workspace as the calling script and the calling script has access to any objects created by the included script.

:clear

Clears all objects from the client's workspace. An error will result if a script clears all objects from the workspace, then subsequently attempts to refer to a cleared object.

:list

Returns a list of the names of all objects in the client's workspace. This is generally only useful as the last instruction in a given script as subsequent return value requests will overwrite the output of this command.

Scripting Examples

Reading data from file (local or network)

This example reads data from an SPC-format file located on a remote computer, reads a PLS model from a .mat file assumed to be the only variable in a file located on the local computer, and applies the model to the data. Finally, it returns the predictions from that application.

```
-----  
data = '//datacollector/datafolder/spectrum1.spc';  
model = 'C:/modelfolder/mymodel.mat';  
pred = data | model;  
pred.prediction  
-----
```

Recall that spaces and linefeeds are not necessary (only the semicolons are necessary) but they are included in the above script to help readability.

Two subsequent calls are then used to issue commands to retrieve first the T^2 value:

```
pred.t2
```

and then the Q (sum squared residuals) values:

```
pred.q
```

Reading data from passed CSV

This example starts by pre-reading a PCA model (when the client first starts), then makes predictions using that model by passing directly to the server in a comma-separated values format. The Q contributions and Hotelling's T^2 contributions are then retrieved.

When the client starts, it sends the single-command script

```
model = 'C:/modelfolder/mymodel.mat';
```

This model will remain in memory as long as the server is running.

Next, the following script is executed each time the client is ready to make a prediction. It passes the seven values expected by the model and makes a prediction.

```
!plain;  
data = '0,100,1242,2320,14,-50,232';  
pred = data | model;
```

Because this script returns no outputs and uses the "plain" response format, the successful application of the model will be indicated by an empty message returned by Solo_Predictor. The client can easily test for any errors by checking for a non-empty return string from the server.

With a successful model application, the client can retrieve the Q contributions:

```
!plain; pred.qcon
```

Again, the script requests a "plain" response format so the Q contributions (which are a row-vector of numbers) are returned in a space-delimited format. Finally, another script is sent to retrieve the T^2 contributions:

```
!plain; pred.tcon
```

The comma-separated return values from this script can also be easily parsed by the client.

Reading data from passed XML

To modify the previous script to pass an XML data structure and include a series of labels with the data, the following script would be used for prediction calls:

```

:plain;
data = '<obj class="dataset">
<data class="numeric"> 0,100,1242,2320,14,-50,232</data>
<label>
<set>
<mode>2</mode>
<name>variable labels</name>
<content>
<sr>tempA</sr>
<sr>Aspeed</sr>
<sr>tempB</sr>
<sr>Bspeed</sr>
<sr>Rate</sr>
<sr>SetPoint</sr>
<sr>Time</sr>
</content>
</set>
</label>
</obj>';
pred = data | model;

```

For more information on the XML format, see `DataSet_XML_Format`.

Wait for file and output to results file

Wait for file provides a simple method to interface to programs which do not have the capability to send data through sockets or an ActiveX interface. In this example, we will assume that a client program has been configured to save data files (in .xy format) to a specified "drop folder". Solo_Predictor will monitor this folder and, when a new file appears, it will load the data into an object named "data" (this name is not configurable) and execute an analysis script we will call "analyze.scr".

For this mode, the following options must be configured:

waitforfile must be "on"

waitfolder must give the full path to the drop folder ("c:/df" in this example)

waitfilespec must be defined as "*.xy" (to ignore all other file types in the folder.)

waitscript must point to the "analyze.scr" with the entire path to the file.

The analyze.scr will contain the steps we want to use to analyze the results, and it will contain the instructions to write the results out to a results file.

analyze.scr

```

modl = '//network-drive/modelfolder/mymodel.mat';
pred = data | modl;
!writefile 'c:/df/results.txt' 'c:/df/template.tmp'

```

The first line loads the model from a .mat file located on a network drive. The second line applies the model to the data loaded from the dropped file. The final line writes the results out to a file named "results.txt" in a folder named "df". The template file would be defined to output any of the results needed from this application. For an example, see the `!writefile` definition in the Script Construction section.

Retrieved from "http://wiki.eigenvector.com/index.php?title=Solo_Predictor_Script_Construction"

- This page was last modified on 18 November 2009, at 20:50.

DataSet XML Format

From Eigenvector Documentation Wiki

This page describes the XML format to construct a DataSet object. The DataSet object is a container for scientific data which permits the storage of numerical values (known as the data) along with the typical associated contextual information.

For the purposes of this appendix, it is important to note that the DataSet object allows the inclusion of one or more sets of textual labels to be associated with each column and/or row of a data matrix. In addition, numerical "axis scale" values can also be associated with each column or row.

It should be noted that the object has a significant amount of flexibility beyond what this document will discuss. For additional information on the fields and construct of the DSO, the user is directed to the object's documentation on the web: [1] (<http://software.eigenvector.com/DataSet/>)

By convention in Solo and PLS_Toolbox, each row of a data table is considered a "sample" (or "observation") and the columns of a data table are the variables measured on each sample. Thus, to create a typical DataSet object (DSO) which can be used to make a prediction, a DSO will be created around a single row of values. An XML construct of a DSO will, therefore, always contain at least a tag to describe the data.

Numerical values for an XML construct of a DSO are given in comma-separated and semicolon-separated format. Commas indicate values on the same row of a matrix (item,item,item); Semicolons indicate row-wise breaks (row; row; row). White space is always ignored.

The basic XML DSO construct consists of the outer object tag with a "class" attribute indicating that the object is a DSO. There are actually two formats for creating a DataSet objects. One uses class="dataset" and is a complete and complicated description of a dataset object. The other uses class="dso" and is much easier to create. We recommend class="dso" for most applications and will not discuss class="dataset" in this document.

The outer tag must contain a <data> tag which will always have the class="numeric" attribute (because data will always be numeric).

```
<obj class="dso">
  <data class="numeric"> 1,2,3,4,5 </data>
</obj>
```

This XML construct would create a simple DSO containing the values 1 to 5 in a row vector. If the DSO being created should contain multiple rows, a semicolon should be used after each row of numbers. Note that all rows **must** contain the same number of elements.

In most cases, it is desirable to associate some contextual information regarding the variables which are being passed to the predictor. This is often expressed as either textual labels, indicating the measured parameter (often giving the name of purpose of the device: "thermocouple A") or numeric axis scale values (often used in spectroscopy, electrochemistry, time-based measurements, etc.) These contextual data will be used by Solo_Predictor to help align new data to a model, verify that the new data has all the expected variables, and replace those which are missing.

To include labels in a DSO, an additional <label> tag must be added to the XML description. The label tag can

contain one or more label "sets" each enclosed in a <set> tag. Each set contains three elements: mode, name, and content. The mode tag indicates the data mode (1=rows/samples, 2 = columns/variables) for which the label set is being defined. The name tag is optional but, if present, indicates a name to associate with this label set. The content tag defines the actual labels for *each element* on the given mode. Each label must be enclosed in its own separate <sr> tag and there must be an appropriate number of tags for the number of columns or rows (whichever mode the labels are being associated with). For example, the following creates the labels "A" through "E" for the five columns of our example data:

```
<obj class="dso">
  <data class="numeric"> 1,2,3,4,5 </data>
  <label>
    <set>
      <mode>2</mode>
      <name class="string">example variable labels</name>
      <content class="string">
        <sr>A</sr>
        <sr>B</sr>
        <sr>C</sr>
        <sr>D</sr>
        <sr>E</sr>
      </content>
    </set>
  </label>
</obj>
```

A label can be added for the sample (first mode) by including an additional <set> tag inside the <label> tag (before or after the <set> tags already included above):

```
<set>
  <mode>1</mode>
  <name>example sample label</name>
  <content class="string">
    <sr>This is my one sample</sr>
  </content>
</set>
```

Although the <content> tag uses the <sr> tags to enclose the string, this is not necessary in this case. Any time a single string value is being created, the <sr> tags can be omitted as can the class attribute. Thus the content tag could have read:

```
<content>This is my one sample</content>
```

Numeric axis scale values can be added using an axisscale tag (note the tag name does not contain a space) with similar content to the label tag. The only difference is that the axisscale property expects a numeric value so the <content> tag is defined with the class="numeric" attribute and the values are supplied as a comma-separated values list. The following defines an axisscale for the variables running from 500 to 508 in steps of 2:

```
<obj class="dso">
  <data class="numeric"> 1,2,3,4,5 </data>
  <axisscale>
    <set>
      <mode>2</mode>
      <name>example axis scale</name>
      <content class="numeric">
        500,502,504,506,508</content>
    </set>
  </axisscale>
</obj>
```

As with labels, note that the number of items defined in the content must match the length (number of elements) of the given mode (columns in this example).

Most of the remaining DSO properties (fields) can be set using similar calls. For example, classes and titles (see the DataSet object documentation for more information on these fields) can be added to the DSO using tags similar to label and axisscale. Titles must have content of class="string" and must contain a single string. Classes can have numeric or string content and must have sufficient elements to match the size of the given mode.

In addition, the include field uses the <set> notation described above and the author, name, description, and userdata fields all use the single-tag notation (as with the data tag where the field name is given with the class attribute and the content within the tag). For example, see below:

```
<obj class="dso">
  <data class="numeric"> 1,2,3,4,5 </data>
  <name class="string">Name for Dataset</name>
  <author class="string">Dataset\'s Author</author>
  <description class="string">
    <sr>Include a multi-line string here</sr>
    <sr>Use as many sr tags as you have lines</sr>
  </description>
</obj>
```

Note the use of the backslash in front of the single quote included in the Author tag. This is only necessary when passing XML through the Solo_Predictor interface. When XML is saved to a file, backslashes are not needed.

```
<obj class="dso">
  <data class="numeric">1,2,3,4,5</data>
  <name>Name for Dataset</name>
  <author>Dataset\'s Author</author>
  <description class="string">
    <sr>Include a multi-line string here</sr>
    <sr>Use as many sr tags as you have lines</sr>
  </description>
  <axisscale>
    <set>
      <mode>2</mode>
      <name>example axis scale</name>
      <content class="numeric">
        500,502,504,506,508</content>
    </set>
  </axisscale>
  <label>
    <set>
      <mode>2</mode>
      <name>example variable labels</name>
      <content class="string">
        <sr>A</sr>
        <sr>B</sr>
        <sr>C</sr>
        <sr>D</sr>
        <sr>E</sr>
      </content>
    </set>
    <set>
      <mode>1</mode>
      <name>example sample label</name>
      <content>This is my one sample</content>
    </set>
  </label>
</obj>
```

Please contact Eigenvector Research for more information on DSO XML format, if needed.

Retrieved from "http://wiki.eigenvector.com/index.php?title=DataSet_XML_Format"

-
- This page was last modified on 15 June 2009, at 20:28.

Script Commands Summary

From Eigenvector Documentation Wiki

Contents

- 1 Import of data into object
- 2 Apply one object to another
- 3 Return request command:
- 4 Output Format commands:
- 5 Write To File command:
- 6 Other Commands
- 7 Comments

Import of data into object

```
obj = 'filename.ext'  
obj = 'fully/qualified/path/filename.ext'  
obj = 'filename.mat?variable'  
obj = '<xml>(PLS_Toolbox XML Format)</xml>'  
obj = '3,4,5'
```

Apply one object to another

- Apply a model object to a data object:

```
result = data|model
```

- Apply a preprocessing object to a data object:

```
result = data|preprocessing
```

- Augment two data objects together:

```
result = data_1|data_2
```

Fields in "result" will depend on the model and object types.

Return request command:

Return value of object or property of object:

```
obj
obj.property
```

Output Format commands:

```
:xml          -set output format to XML
:plain        -set output format to plain text
:html         -set output format to HTML
```

Write To File command:

Create output file using a template file for formatting

```
:writefile 'output.ext' 'template.tem' -append
```

Optional flag "-append" will append to end of output.ext instead of replacing the file. If 'template.tem' filename is omitted, a template file named 'output.tem' is expected in the same folder as the output file.

The Template file can contain static text and replacement fields using square brackets around return-request-style codes, for example:

```
"the T-squared value is [pred.t2]"
```

replacement fields can also contain a formatting code using standard fprintf formatting:

```
"the Q value is [%0.2e,pred.q]"
```

Note that the template file **must** have the .tem extension.

Other Commands

```
:include 'myscript.txt'
```

Executes the script in myscript.txt.

Included scripts are executed at the point in the calling script where the include statement occurs. If any errors happen during myscript.txt, all script execution will stop at the include statement. Commands after the include can use objects created within the script (included scripts share the same workspace).

```
:help          Returns list of valid commands
:list          Returns list of current objects
:clear         Clears all objects
```

Comments

```
//Comment ;  
\#Comment ;  
%Comment ;
```

Any text following one of the comment characters shown above (and before any end-of-line character) is ignored. This allows users to include comments into their scripts to make them more human-readable.

Retrieved from "http://wiki.eigenvector.com/index.php?title=Solo_Predictor_Script_Commands_Summary"

- This page was last modified on 15 June 2009, at 20:54.

Example Connection Code

From Eigenvector Documentation Wiki

Contents

- 1 C# - Socket Connection
- 2 VB – ActiveX
- 3 Matlab – Socket Connection

The following provides some standard code pieces which can be used to make connections to Solo_Predictor. These examples may not be appropriate for all applications (e.g. none of these examples are asynchronous calls. They all 'block' the program execution until Solo_Predictor returns a result.) In addition, additional error checking and input/output parsing is required in most cases. Lots of other code examples can be found on the web by searching for sockets and the language of interest.

C# - Socket Connection

The following C-sharp example expects three variables: ServerIP as a string indicating the server's IP address, Port as an integer giving the server's port number, and command as a string command to send. It returns Solo_Predictor's output in a variable called outputString. This code requires the following "using" declarations:

```
using System;  
using System.Net.Sockets;  
using System.ComponentModel;
```

Note that no object or function declarations are given in this code but would generally be required. This function also sets the ReceiveTimeout property on the socket to 2000 milliseconds. This setting can be adjusted as required by the application. The total time of most calls to Solo_Predict is 1/2 second or less (testing on a moderately featured system in 2007 indicated application of a PLS model to a 200 point vector took an average of 0.2 seconds including I/O overhead.)

```

TcpClient socketForServer;
try
{
    //make connection to server
    socketForServer = new TcpClient(ServerIO,Port);
}
catch
{
    outputString = "ERROR: Failed to connect to server";
    return;
}
//get stream and stream reader/writer
NetworkStream networkStream = socketForServer.GetStream();
System.IO.StreamReader streamReader = new System.IO.StreamReader(networkStream);
System.IO.StreamWriter streamWriter = new System.IO.StreamWriter(networkStream);
try
{
    //send command to server
    streamWriter.WriteLine(command);
    streamWriter.Flush();
    //wait for and retrieve response
    string outputString;
    socketForServer.ReceiveTimeout = 2000;
    outputString = streamReader.ReadToEnd();
}
catch
{
    outputString = "ERROR: Exception reading from Server";
}
networkStream.Close(); // tidy up

```

VB – ActiveX

See the EigenvectorTools Visual Basic example.

Matlab – Socket Connection

This code example runs in Matlab and makes use of Java commands to create a socket connection, send a message, and retrieve the response. It assumes that an input message exists in the variable msg as a string command to send and the server ip and port are in the variables srv and port.

```

import java.io.*;
import java.net.*;

%Create socket connection and socket reader and writers
clientSocket = java.net.Socket(srv,port);
iStream_client = clientSocket.getInputStream;
iReader_client = InputStreamReader(iStream_client);
outStream_client = clientSocket.getOutputStream;

%create buffers for socket
clientOut = PrintWriter(outStream_client, true);
clientIn = BufferedReader(iReader_client);

%send message to Solo_Predictor
clientOut.println(java.lang.String(msg));
clientOut.flush;

%wait for reply ready
starttime = now;
while ~clientIn.ready
    if (now-starttime)>60/60/60/24;
        error('No response from server')
    end
end

%read in reply and store in cell array
rcv = {};
while clientIn.ready;
    rcv{end+1} = char(readLine(clientIn));
end

%concatenate string reply with linefeeds
if length(rcv)>1;
    rcv = sprintf('%s\n',rcv{:});
else
    rcv = rcv{1};
end

```

Retrieved from "http://wiki.eigenvector.com/index.php?title=Solo_Predictor_Example_Connection_Code"

- This page was last modified on 15 June 2009, at 21:08.