



Model_Exporter Users Manual and Technical Notes

Table of Contents

Table of Contents	2
Introduction.....	3
System Requirements.....	4
Matlab-Based Exporter Requirements.....	4
Stand-Alone Exporter Requirements	4
Requirements for Using Exported Models	4
Supported Methods	5
Exporting a Model	6
Exporting from PLS_Toolbox and MATLAB.....	6
Exporting from Solo	6
MATLAB M-file Format Description	7
Use of Exported M-files.....	7
Input Data.....	7
Returned Results	8
Creating Functions from Exported Models.....	8
Tcl-file Format Description and Use	9
Use of Exported Tcl-files.....	9
Input Data.....	9
Returned Results	10
XML Format Description	11
Numerical Matrix Definitions.....	11
XML Structure	11
Requirements for XML Interpreters	13
Managing of Constants and Variables	13
Script Execution.....	14
Mathematical Operation Requirements	14
Script Execution Requirements.....	15
Returned Results	15
Requirements for XML Writer	16

Introduction

Model_Exporter from Eigenvector Research, Inc. converts models created within the PLS_Toolbox or Solo chemometrics modeling environments into an interpretable format for use outside of these products. These exported models can be used with a user-supplied interpreter to make predictions on new data. One prediction engine is supplied as an example.

Model_Exporter takes as input a standard model structure created in PLS_Toolbox or Solo and outputs the model into one of three formats: an XML file (executable by a user-supplied external parser), an m-file (executable in MATLAB – separately distributed by Mathworks, Inc – without any additional toolboxes) or a Tcl file (executable in a Tcl interpreter or in the Symbion software package – by Symbion Systems, Inc.).

The exported model requires very few resources to be executed. Specifically, it requires floating-point numerical calculations, a small amount of memory, and the overhead resources required by the specific interpreter.

This documentation describes the use of the Model_Exporter, the use of exported m- and Tcl-formatted files as well as to help in the design of external XML parsing engines. One example engine is supplied for the PHP language (often used for web-page scripting predictions; see <http://www.php.net> for more information on PHP). Additional engines may be available - Contact Eigenvector Research, Inc. for more information.

System Requirements

Model_Exporter can be executed from either the MATLAB computational environment (Mathworks, Inc., Natick, MA), or Solo (Eigenvector Research, Inc., Wenatchee, WA). Model_Exporter converts models created by PLS_Toolbox 3.5 or higher or Solo 4.0 or higher.

Matlab-Based Exporter Requirements

For execution of Model_Exporter within the MATLAB environment, the following is required:

- Matlab 6.5 or higher

- 256 MB RAM (recommended – less may be required)

Solo-Based Exporter Requirements

For execution of the Model_Exporter, the following is recommended

- Solo 4.1 or higher

- Windows 2000, XP, 2003 server, Vista, or MAC OS X (intel)

- 200 MB Disk Space (for installation; some models may require additional space)

- 256 MB RAM (recommended – less may be required)

Requirements for Using Exported Models

The requirements to execute an exported model vary depending on the interpreter used, the number of variables in the modeled data, and the complexity of the model (i.e. the number of factors/components included in the model and the types of preprocessing used).

Memory requirements depend on the precision required for the application, the number of variables in the data and the total number of factors in the model. For example, a model working on 10,000 variables and 5 factors would require around 1MB for double-precision calculations and 500KB single-precision calculations.

The software which executes the specific file formats may have additional requirements. See the file format description sections later in this manual for where to locate model execution details.

Supported Methods

Model_Exporter supports the following model types:

PCA – Principal Components Analysis model

PLS – Partial Least Squares regression model

PCR – Principal Components Regression model

and preprocessing methods:

Absolute value	Autoscale	Derivative (SavGol)	Detrend
GLS weighting	Log10	Mean center	Median center
Normalize	OSC	Smooth (SavGol)	SNV
Sqrt Mean Scale	MSC		

Model_Exporter does not support replacement of missing values (values must be measured for all variables).

Exporting a Model

Exporting from PLS_Toolbox and MATLAB

Model_Exporter is easily called from the MATLAB environment. After adding the Model_Exporter to the MATLAB path, a model can be exported by simply calling the `exportmodel` function, passing the model structure itself, and an optional input specifying the file name and type to which the exported model should be written. When filename is omitted, Model_Exporter will prompt for a filename, file type, and location.

```
exportmodel(modelstructure, filename)
```

Model_Exporter is also accessible from the PLS_Toolbox through the Analysis GUI. With the model to export loaded into the Analysis GUI, go to the **File / Export Model / To Predictor...** menu and select the file type to export from the flyout menu.

Exporting from Solo

When installed with the stand-alone Solo software, a model is exported from the Analysis GUI. With the model to export loaded into the Analysis GUI, Go to the **File / Export Model / To Predictor...** menu and select the file type to export from the flyout menu.

MATLAB M-file Format Description

Use of Exported M-files

The m-files output by the Model Exporter are stand-alone. That is, they can be run by the MATLAB computational environment (available from Mathworks, Inc., www.mathworks.com) without any additional toolboxes.

For maximum flexibility, an exported model is written as a MATLAB script which expects only to find a variable named `x` in its workspace. This variable provides the input data to which the model should be applied. It is important to note that the variable `x` will be modified by the script and, thus, the caller should *not* expect the variable to remain unchanged. See "Creating Functions from Exported Models", below, for more information on how to isolate the script and call it as a function. (Those unfamiliar with MATLAB scripts and functions should read the MATLAB documentation describing these concepts and the associated "variable scope" documentation.)

Input Data

The expected length (number of elements) and contents of the input `x` vector are defined in the comments and initial sections of the exported model script. The script, as exported, does not use this information to perform any validity testing on the input variable. This information is only provided to indicate to the user what type of data is expected.

The example below shows the part of an exported model which indicates the expected data size and associated context information. This particular model expects input data of ten variables as a row vector (as described by `inputdata.size`). The labels of these ten variables are specified in the string array `inputdata.label`. As there was no axis scale information in this particular data, the `inputdata.axis scale` value is empty.

```
inputdata.size = [ 1 10 ];  
inputdata.axis scale = [ ];  
inputdata.label = ['Fe'; 'Ti'; 'Ba'; 'Ca'; 'K '; 'Mn'; 'Rb'; 'Sr'; 'Y '; 'Zr'];
```

The user can make use of this information to assure the data being passed to the model is correct. Again, as written, the script provides no testing. Incorrect data sizes will be indicated by a runtime error when executing the script.

Returned Results

The results available from a model prediction will be present as variables in the script's workspace. The user is responsible for making use of these variables as needed. The following list specifies the supported results which may be of interest to the user.

<code>scores</code>	Scores for each component as a row vector.
<code>Tcon</code>	Variable contributions to T^2 as a row vector.
<code>Xhat</code>	Model estimate of the data (\hat{x}) as a row vector.
<code>Qcon</code>	Q residuals contributions (x residuals) as a row vector.
<code>T2</code>	The Hotelling's T^2 as a scalar value.
<code>Q</code>	The sum squared x residuals (Q value) as a scalar value.

Regression models (PCR and PLS) return the following additional value:

<code>yhat</code>	Model prediction for y (predicted y value) as a scalar value or vector.
-------------------	---

Creating Functions from Exported Models

Although the exported model is written as a script which would normally operate in the base workspace of MATLAB, the user can also wrap the script into a function by simply adding a standard function definition line at the beginning of the script file. This function definition would require only one input, `x`, and could output any of the variables which are present after the script's execution. An example would be:

```
function [scores,Q,T2,Qcon,Tcon] = mymodel(x)
```

This function definition returns the vectors: `scores`, `Qcon`, and `Tcon`, as well as the scalar values: `Q` and T^2 to the caller. A function wrapper also keeps the input variable `x` from being modified outside the function. This approach tends to be safer than a script, but is not implemented by default in order to provide the widest flexibility to the user.

Tcl-file Format Description and Use

Use of Exported Tcl-files

The tcl-files output by the Model Exporter can be run by either a stand-alone Tcl parser (for example see the "Batteries Included" ActiveTcl Distribution <http://www.tcl.tk/software/tcltk/>) or by Symbion (available from Symbion Systems, Inc., www.gosymbion.com). When run in a stand-alone Tcl parser, the La package for matrix support is required (available free from: <http://www.hume.com/la/>)

For maximum flexibility, an exported model is written as a Tcl script which expects only to find a variable named x in its workspace. This variable provides the input data to which the model should be applied. It is important to note that the variable x will be modified by the script and, thus, the caller should *not* expect the variable to remain unchanged.

Input Data

The expected length (number of elements) and contents of the input x vector are defined in the comments and initial sections of the exported model script. The script, as exported, does not use this information to perform any validity testing on the input variable. This information is only provided to indicate to the user what type of data is expected.

The example below shows the part of an exported model which indicates the expected data size and associated context information. This particular model expects input data of ten variables as a row vector (as described by `inputdata.size`). The labels of these ten variables are specified in the string array `inputdata.label`. As there was no `axisscale` information in this particular data, the `inputdata.axisscale` value is empty.

```
# inputdata.size = [ 1 10 ];
# inputdata.axisscale = [ ];
# inputdata.label = ['Fe';'Ti';'Ba';'Ca';'K '; 'Mn';'Rb';'Sr';'Y
';'Zr'];
```

The user can make use of this information assure the data being passed to the model is correct. Again, such testing is not provided by the script as written. Incorrect data sizes will be indicated by a runtime error when executing the script.

Returned Results

The results available from a model prediction will be present as variables in the script's workspace. The user is responsible for making use of these variables as needed. The following list specifies the supported results which may be of interest to the user.

<code>scores</code>	Scores for each component as a row vector.
<code>Tcon</code>	Variable contributions to T^2 as a row vector.
<code>Xhat</code>	Model estimate of the data (\hat{x}) as a row vector.
<code>Qcon</code>	Q residuals contributions (x residuals) as a row vector.
<code>T2</code>	The Hotelling's T^2 as a scalar value.
<code>Q</code>	The sum squared x residuals (Q value) as a scalar value.

Regression models (PCR and PLS) return the following additional value:

<code>yhat</code>	Model prediction for y (predicted y value) as a scalar value or vector.
-------------------	---

XML Format Description

Numerical Matrix Definitions

The XML format utilizes custom tags to define various parts of the model. For some tags, the content is a vector or matrix of values. In these cases, a comma character delineates different column elements and semicolon indicates the end of a matrix row and the beginning of the next. All white space is ignored. If a given matrix contains only one row, it is described as a "row vector". A matrix with a single column is described as a "column vector". Orientation of such vectors is critical to the mathematical operations and must be parsed appropriately.

XML Structure

The XML file will consist of a top level <model> tag which will contain an <information> tag, an <inputdata> tag, and one or more step segments, each wrapped in a separate <step> tag.

- <model>**
- <information>** General information on the encoded model.
 - <source>** Text description of file source (EVRI Model_Exporter).
 - <modeltype>** Standard model method acronym (PCA, PLS, etc).
 - <description>** Text description of model including preprocessing, data size(s), and number of components. Each row of this multi-row string is delineated by <sr> (string row) tags.
 - <datasource>** Information block of modeled calibration data. <datasource> is a multi-cell table format. There will be one column of information for each block of data required by the given modeltype (e.g. PCA requires 1 block, PLS requires 2). Each <td> tag will contain a number of sub-fields describing the data used for the given block. Informational only, sub-fields may change.
- </information>**
- <inputdata>** Specific requirements for input data including the following information:
 - <size>** Numeric class row vector describing the size expected for the input data (x). The first element of the vector gives the expected number of rows, the second is the expected number of columns.
 - <axisscale>** Numeric class row vector providing the expected axisscale of the input values. The actual values stored in the axisscale vector are completely dependent on the application and the analytical method used and may be empty.
 - <label>** Strings (delimited by <sr> sub-tags) defining the names of the variables expected in the input data (x). The names are

dependent on the application and the analytical method used and may be empty.

</inputdata>

<step>

Repeated tag for each step required for making a prediction using this model. Will contain the following sub-fields:

<sequence>

Numeric class single value indicating the order in which this step should be performed. The steps are generally included in the XML file in sequence-order (sequence 1 will be the first step in the file), but this field can be used to assure in-order processing of steps.

<description>

String class description of the step (informational only)

<constants>

Contains information on constants required by this step. Each constant is defined as a sub-tag herein. The name of the constant is the sub-tag name and will contain a matrix (or vector) of values to use for the given constant. See below for more information.

<script>

One or more rows of strings describing the mathematical operations to perform this step. When more than one mathematical operation is to be performed, each will be given in a separate string row **<sr>** tag. However, these can be ignored. Each mathematical operation will be terminated with a semicolon.

</step>

(Additional <step> tags located here...)

</model>

See the provided files "pcaexample.xml" and "plsexample.xml" for full examples of the XML structure.

Requirements for XML Interpreters

To execute each of the <step> segments contained in the XML file, an interpreter must be able to parse the constants defined into matrices and be able to execute the script commands. The following give the specifications for an interpreter.

Managing of Constants and Variables

- The interpreter must maintain a "workspace" of stored constants and variables in which the matrices can be accessed by a variable name (specified by the tag in which the given constant was read, for example: <s class="numeric" size="[1,1]">4</s> would define a constant "s" which was equal to the scalar value 4).
- Constants are NOT case sensitive and any interpreter must be written to consider the upper or lower case variables as the same.
- "Constants" are just pre-defined variables. Although every effort will be made to avoid changing these values, it is NOT a rule that these "constants" cannot be changed – scripts may modify and overwrite these values. They are called "constants" because they are initially defined by the model.
- The enclosing tag for the constant will define the class of the constant (in this application, constants will always be "numeric") and will also define the size of the constant using the attribute 'size'. For example, <s class="numeric" size="[1,5]"> defines that the enclosed constant will be a row vector (1 row) of 5 elements (5 columns).
- Prior to the execution of the script(s), the XML interpreter must place a variable named "x" (lower-case) in their workspace. This variable must contain the data to which the model should be applied. The value of "x" will be modified by the script so, following initial assignment, no alteration of this variable should be done outside what is specified by the script.
- All constants/variables must be retained for the entirety of a given step. In many cases, the variables remaining in the workspace will contain results of interest to the caller and, therefore, all workspace values should be retained. The variable "x" must always be present.

Script Execution

The following lists define the script commands which must be supported by the interpreter (scripts may contain only these commands). When applicable, the Matlab operator corresponding to the given function is given. Interpreters do *not* need to interpret these operators. They will never be used in any script and are provided here only for reference.

Single Input Functions: $C = \text{function}(A);$

abs	Absolute Value	Removal of sign of elements
log10	log (base 10)	Base 10 logarithm of elements
transpose	transpose array	Exchange rows for columns (')

Double Input Functions $C = \text{function}(A,B);$

plus	Plus	Addition of paired elements (+)
minus	Minus	Subtraction of paired elements (-)
mtimes	Matrix multiply (dot product)	Dot product of matrices (*)
times	Array multiply	Multiplication of paired elements (.*)
power	Array power	Exponent using paired elements (.^)
rdivide	Right array divide	Division of paired elements ./)

Mathematical Operation Requirements

- All mathematical operations are expected to be performed using **signed, single precision** numbers.
- With the exception of mtimes (dot product), all operations are "element-by-element". That is, the two matrices passed will be equal in size (see scalar exception below) and the mathematical operation is performed between each element of matrix A and its corresponding element in matrix B. The output matrix C is always the same size as A and B.
- Scalar Exception (except mtimes): A or B may be a scalar even if the other isn't. In this situation, the scalar input must be interpreted as an appropriately-sized matrix containing all the same value.
- mtimes (dot product) is performed using the standard linear-algebraic dot-product operation. In generic terms, the input matrix A will contain m rows and k columns, the input matrix B will contain k rows and n columns and the output matrix C will contain m rows and n columns. The following equation is used to calculate each element of the C matrix (loop for $i = 1$ to m and for $j = 1$ to n):

$$C_{i,j} = A_{i,1}B_{1,j} + A_{i,2}B_{2,j} + A_{i,3}B_{3,j} + \dots + A_{i,k}B_{k,j}$$

Subscripts indicate the row and column indexing (respectively) into the given array. When either A or B is a scalar, the mtimes operation should be handled as a "times" operation. That is, the operation becomes an element-by-element multiplication where each element of the matrix input is multiplied by the scalar value and C is the same size as the input matrix.

Script Execution Requirements

- The format for a single script command is:

```
C = function(A,B);
```

where *function* is one of the above functions, A and B are the pre-defined constants / variables to use as input to *function*, and C is the output. Input B will be omitted for functions which require only one input. Each command of the script will end in a semi-colon ";". All commands must be performed in the order in which they appear in the script.

- The expected size, axis scale, and labels associated with x will be stored in the < sourcedata > tab (if any exist). These values can be used by an XML interpreter to verify the data being analyzed.
- Constants are NOT case sensitive and any interpreter must be written to consider the upper or lower case variables as the same.

Returned Results

The results returned by a model prediction will be present as variables in the interpreter's workspace upon completion of the XML parsing. The following list specifies the supported results which may be of interest to the caller.

scores	Scores for each component as a row vector.
Tcon	Variable contributions to T^2 as a row vector.
Xhat	Model estimate of the data (\hat{x}) as a row vector.
Qcon	Q residuals contributions (x residuals) as a row vector.
T2	The Hotelling's T^2 as a scalar value.
Q	The sum squared x residuals (Q value) as a scalar value.

Regression models (PCR and PLS) return the following additional value:

yhat	Model prediction for y (predicted y value) as a scalar value or vector.
------	---

Requirements for XML Writer

The following rules are to be followed by the script creation algorithm of Model_Exporter. These rules may be of interest to script interpreters, but should not have any critical impact on interpreter design.

- Nesting of functions is *not* allowed. Functions can only take variables or pre-defined constants as input.
- NO iterative processes are supported. All scripts must be straight-through executing (no control structures such as "ifs", "while", etc are supported.)
- Missing data replacement is not supported.
- As of version 1.0 of this product, only variables or pre-defined constants may be used in a function. No "in-line" constants may be used. For example:

```
C = minus(A, 1);
```

is invalid because the constant "1" has to be pre-defined. This command should instead be written where the "1" is pre-defined as a constant and the name of that constant is used.

- Variables are NOT case sensitive and any interpreter must be written to consider the upper or lower case variables as the same. Note, however, that the Matlab output of this function will be case-sensitive code so the scripts should try to be consistent in case, even if other interpreters won't care.