PEAXACT Application Server User Manual

Version 3.7.2
2015-06-23

S•PACT GmbH
Burtscheider Str. 1
52064 Aachen
Germany

phone: +49 241 9569 9812
fax: +49 241 4354 4308
e-mail: support@s-pact.de

CONTENTS

1 QUICK START

1.1 What is PEAXACT Application Server?

The PEAXACT Application Server gives third-party applications access to PEAXACT analysis methods by means of an application programming interface (API). The API is available as:

- COM (Component Object Model)
- .NET assembly

Any application supporting one of these standards will be able to programmatically integrate PEAXACT as a back-end analyzer for spectroscopic or chromatographic data.

In addition, the Application Server provides customized, ready-to-use interfaces for:

- OPUS Process from Bruker
- HoloPro from Kaiser Optical Systems

1.2 Getting Help

User Manual

This user manual documents a certain version of the PEAXACT Application Server. You can find the version number and publication date on the title page.

We are continuously working on improving the manual. The latest document version is distributed as PDF file with each PEAXACT software update. The file is located in subdirectory `help` of the PEAXACT installation directory.

Technical Support

The Technical Support can be contacted in different ways:

- E-mail to support@s-pact.de
- Web form at http://www.s-pact.de/support

> Note: A subscription of S•PACT Software Maintenance Service (SMS) is required to be eligible for technical support. The first year of SMS is included with new PEAXACT product licenses.

Blog

The PEAXACT Blog was launched as a free source of information complementary to the user manual. It contains tutorials, how-tos, and tips & tricks.
See: http://www.s-pact.de/blog

1.3 Installation

1.3.1 System Requirements

- Microsoft Windows XP SP2 or later (32 bit or 64 bit)
- Intel or AMD x86 or x64 CPU with SSE2 support (2 GHz recommended)
- 1 GB of disk space (2 GB recommended)
- 1 GB RAM (2 GB recommended)

1.3.2 Licensing

PEAXACT software is furnished under a license agreement. The software may be used only under the terms of the license agreement.

The PEAXACT Application Server can be installed and operated on a given number of designated computers, provided it is only operated locally (i.e. not remotely). The number of simultaneous users is not limited. For the full and legally valid conditions please refer to the license agreement document.

1.3.3 Installation

Step 1: Before You Install

- Make sure your computer fulfills the system requirements.
- When upgrading an existing installation, visit http://www.s-pact.de/peaxact/whatsnew and read the upgrade notes and compatibility considerations.
- Make sure you have administrator privileges to perform the installation.
- Make sure your license is valid for the major version you want to install. If you do not have a license yet, you can get a free trial license or purchase a license after installation.

Note: The PEAXACT version is a concatenation of three numbers `<major version>.<minor version>.<maintenance version>`, e.g. `3.7.0`.

Step 2: Install PEAXACT

- Download the latest PEAXACT Installer from http://www.s-pact.de/downloads

Note: The installer's filename is `peaxactInstaller_<version>_<arch>.exe` `<version>` is the version number; `<arch>` is the software architecture (`win32` or `win64`). PEAXACT is available for 32 bit and 64 bit Windows platforms. The 32 bit version also runs on 64 bit platforms, but not vice versa.

Note: Only one installation of PEAXACT can exist at a time. Installing a newer version will update the existing installation automatically. Installing a different architecture or older version requires uninstallation of the existing version first.

Online Installation (Web Installation)

- If you are going to install PEAXACT on a computer which is connected to the internet, you do not need to download any additional files.
- Run the PEAXACT Installer and follow the setup instructions. Additional runtime packages are downloaded and installed automatically if detected missing.

Offline Installation

- If you are planning to install PEAXACT on a computer without internet access, you have to download additional runtime packages in advance from http://www.s-pact.de/peaxact/runtime
- Make sure to download runtime packages for the same architecture as the PEAXACT installer (32 bit or 64 bit)
- Save all installer files without renaming them to one folder on your hard drive / flash drive.
- Run the PEAXACT Installer file from this folder and follow the setup instructions. Runtime packages are installed automatically if detected missing.

Step 3: After Installation

- After a new product installation, continue with License Activation.
- After upgrading an existing installation, check the upgrade notes at http://www.s-pact.de/peaxact/whatsnew for further upgrade steps.

1.4 License Activation

License activation involves loading a valid license file. If you already have a license file, go ahead to step 3.

Step 1: Find out the computer's Host ID

Note: This step is required for purchased licenses only! For free licenses, proceed with step 2.

For purchased licenses, activation associates the use of PEAXACT with designated computers by means of a Host ID. The Host ID is a MAC address (format xx-xx-xx-xx-xx-xx) or the serial number of volume c (format xxxx-xxxx) of the computer on which PEAXACT is installed.

- Click the Windows start menu and select
  Programs > PEAXACT > Activate PEAXACT Application Server
- Wait until the License Activation Dialog is displayed
- Take the Host ID from the dialog window, then click Cancel.

- If you purchased a license for multiple computers, get one Host ID for each computer.

Note: You can also type `getmac` at the command prompt and use the first MAC address as Host ID.

Step 2: Request license file

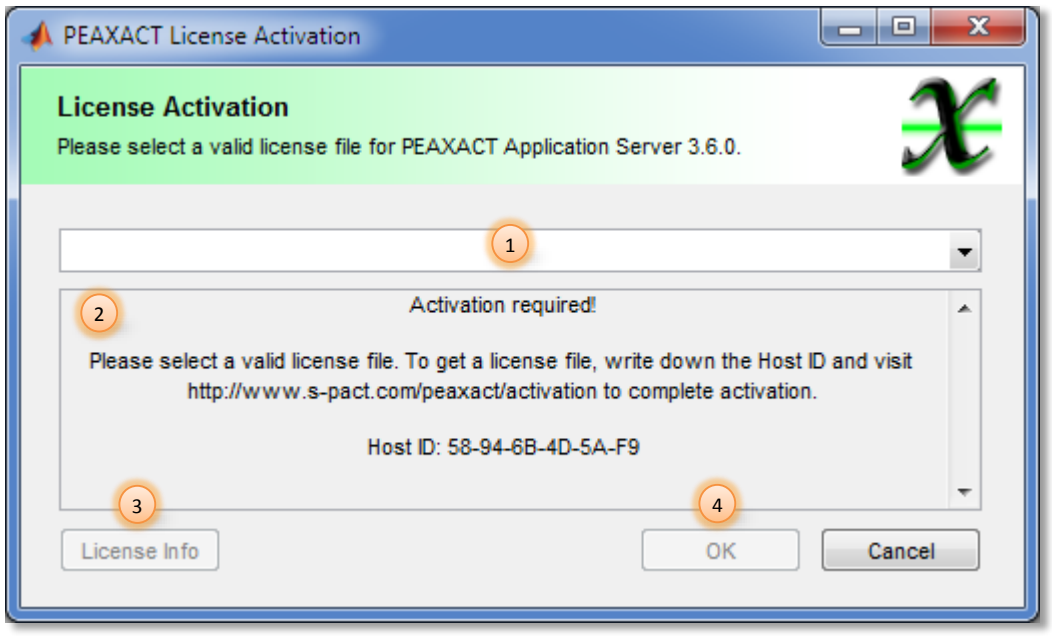
- Visit http://www.s-pact.de/peaxact/activation and use the web form to request a license file.

Step 3: Activate license

Licenses can be activated either programmatically using the `setLicense()` method, or interactively using the License Activation Dialog. While `setLicense()` activates a license temporarily, the License Activation Dialog activates a license permanently by storing the license filename in the Windows registry. The following applies to the License Activation Dialog.

- Click the Windows start menu and select
  Programs > PEAXACT > Activate PEAXACT Application Server
- Wait until the License Activation Dialog is displayed

Note: The License Activation Dialog will also be shown if the Application Server is accessed without a valid license.



License Activation Dialog

(1) License selection
(2) Status of activation
(3) Additional license information
(4) Apply and close

- Choose **Import License...** from the list (1) to browse for a valid license file. If the license is valid the license file is copied to the license directory.

- Once a valid license is selected, you can click on the **License Info** button (3) to learn more about the license or on the **OK** button (4) to accept the selection.

Per-machine license vs. per-user license
If you perform the activation with administrator privileges, licenses will be activated per-machine, i.e. for all Windows users. Otherwise, licenses will be activated per-user, i.e. for the logged on user. Per-machine licenses take precedence over per-user licenses. Once a per-machine license is activated the License Activation Dialog gets locked for regular users.

1.5 Before You Start

1.5.1 COM Component

Before you access the COM API for the first time you should test whether everything is installed correctly by running a ***diagnosis program***. Click the Windows start menu and select **Programs > PEAXACT > Diagnosis Tool for PEAXACT Application Server**.

The diagnosis program performs some tests and suggests possible solutions in case of problems. You have to fix all problems before you can use the COM API. Typical problems include:

- MATLAB Compiler Runtime (MCR) is not installed correctly
- Required DLL files are not registered correctly
- Platform-dependent problems (e.g. running 32 bit software instead of 64 bit)

You could run the diagnosis program at any time to check whether the interface still works correctly and to reveal possible errors.

Note: During the test you may be prompted to activate a license.

1.5.2 .NET Component

The .NET component is a design-time assembly you would compile and link against when building your own managed assemblies. Before you can use it you need to reference the assembly in you Visual Studio project. In Visual Studio, right-click on a project, for example, and click "Add References...". The assembly file is located at

```
INSTALLDIR\DLL\.NET\peaxact.dll
```

You also need to reference the `MWArray` assembly (located in the same directory) which defines the `MWArray` data type. See Section 3.4.2 for further details.

Note: The .NET component requires Microsoft .NET Framework version 2.0 or later to be installed on your computer.

2 INPUT AND OUTPUT FILES

Note: This manual uses the placeholder `APPDATADIR` to reference the directory for user-specific application data. In Windows XP it typically is

```
%UserProfile%\Local Settings\Application Data\
S-PACT\PEAXACT Application Server
```

Since Windows Vista, it is

```
%LocalAppData%\S-PACT\PEAXACT Application Server
```

2.1 License File

The ***license file*** contains information about a license, e.g. the licensed release version. The license can be set using the `setLicense()` method. If no valid license is set, the Windows registry is searched for a license when required. If still no valid license can be found, the user is prompted to select a valid license file.

File Extension

`*.lic` PEAXACT License file

License Filenames in the Windows Registry

```
HKEY_LOCAL_MACHINE\S-PACT\PEAXACT Application Server\licenseSource
```

takes precedence over:

```
HKEY_CURRENT_USER\S-PACT\PEAXACT Application Server\licenseSource
```

License Directory

```
%ProgrammData%\S-PACT\PEAXACT Application Server
```

2.2 Log File

By default, PEAXACT writes information messages, warnings and errors to a ***log file***. The verbosity of the log file can be changed using the `setLogger()` method. If you have multiple applications using the Application Server, you should also use the `setLogger()` to change the default filename.

Default Directory and Filename

```
APPDATADIR\peaxactAppServer.log
```

3 APPLICATION PROGRAMMING INTERFACE (API)

3.1 Introduction

The PEAXACT Application Server is developed in MATLAB and is compiled with the MATLAB compiler to either a COM DLL or a .NET assembly DLL. Despite the different software architecture, both DLLs expose identical classes and methods. All methods are fully documented in subsection Class Reference using MATLAB function signatures. You will be able to translate these function signatures to COM or .NET function calls with the information from subsections Calling Conventions and Data Conversion Rules. You may also learn from Programming Examples.

3.2 Calling Conventions

This Section describes how to translate MATLAB function signatures to COM and .NET function signatures. A detailed function documentation can be found in Section Class Reference.

3.2.1 COM Component

The following tables show the mapping of MATLAB function signatures to IDL code and exemplarily to Microsoft Visual Basic 6.

Functions with inputs only (no output)

Signature	Sample
MATLAB	`function foo(X1, X2, ...)`

IDL	`HRESULT foo([in] VARIANT X1,`
	`  [in] VARIANT X2,`
	`  .`
	`  .`
	`  );`

VB 6	`Sub foo(X1 As Variant, _`
	`  X2 As Variant, _`
	`  .`
	`  .`
	`  )`

VB 6 Example	`Dim com As Object`
	`Dim X1 As Variant`
	
	`' assign values`
	`Set com = CreateObject("myComponent.myClass")`
	`X1 = True`
	
	`' provide all inputs`

Signature	Sample

	`Call com.foo(X1, "test")`
	`' omit some inputs`
	`Call com.foo(X1, Null  )`

The function inputs appear in the same order as they do on the right side of the MATLAB function. All inputs are tagged as `[in]` parameters.

> Note: In MATLAB, all inputs to functions are optional and may be present or omitted from the function call. However, in IDL, function signatures are stricter. You need to pass Null (or an equivalent) in order to omit input arguments from the function call.

Functions with outputs

Signature	Sample
MATLAB	`function [Y1, Y2, ...] = foo(X1, X2, ...)`

IDL	`HRESULT foo([in] long nArgOut,`
	`[in,out] VARIANT* Y1,`
	`[in,out] VARIANT* Y2,`
	`.`
	`.`
	`[in] VARIANT X1,`
	`[in] VARIANT X2,`
	`.`
	`.`
	`);`

VB 6	`Sub foo(nArgOut As Long, _`
	`Y1 As Variant, _`
	`Y2 As Variant, _`
	`.`
	`.`
	`X1 As Variant, _`
	`X2 As Variant, _`
	`.`
	`.`
	`)`

VB 6 Example	`Dim com As Object`
	`Dim X1 As Variant`
	`Dim Y1 As Variant, Y2 As Variant`
	
	`' assign values`
	`Set com = CreateObject("myComponent.myClass")`
	`X1 = True`
	
	`' provide all inputs and outputs`
	`Call com.foo(2, Y1, Y2  , X1, "test")`
	`'omit some outputs`
	`Call com.foo(1, Y1, Null, X1, "test")`

The first argument `nArgOut` is an `[in]` parameter of type `long`. It is the number of requested output arguments. `nArgOut` could be smaller than the total number of possible output arguments in which case MATLAB returns Null for all arguments > `nArgOut`.

Following the `nArgOut` parameter, the outputs are listed in the order they appear on the left side of the MATLAB function, and are tagged as `[in,out]`, meaning that they are passed (by reference) in both directions.

> Note: In MATLAB, all outputs from functions are optional, and may be present or omitted from the function call. However, in IDL, function signatures are stricter. You can use the `nArgOut` parameter to request a certain number of output arguments and pass Null for output arguments > `nArgOut` in order to omit them from the function call.

Functions with mutable inputs and outputs

Signature	Sample
MATLAB	`function [Y1, Y2, ..., varargout] = foo(X1, X2, ..., varargin)`

IDL	`HRESULT foo([in] long nArgOut,`
	`[in,out] VARIANT* Y1,`
	`[in,out] VARIANT* Y2,`
	`.`
	`.`
	`[in,out] VARIANT* varargout,`
	`[in] VARIANT X1,`
	`[in] VARIANT X2,`
	`.`
	`.`
	`[in] VARIANT varargin);`

VB 6	`Sub foo(nArgOut As Long, _`
	`Y1 As Variant, _`
	`Y2 As Variant, _`
	`.`
	`.`
	`varargout As Variant, _`
	`X1 As Variant, _`
	`X2 As Variant, _`
	`.`
	`.`
	`varargin As Variant)`

VB 6 Example	`Dim com As Object`
	`Dim X1 As Variant`
	`Dim varargin(1 To 2) As Variant`
	`Dim Y1 As Variant, Y2 As Variant, Y3 AS Variant, Y4 As Variant`
	`Dim varargout As Variant`
	
	`' assign values`
	`Set com = CreateObject("myComponent.myClass")`
	`X1 = True`
	`varargin(1) = 1.5`
	`varargin(2) = "option A"`
	
	`' provide all inputs and outputs`
	`Call com.foo(4, Y1, Y2,  varargout, X1, "test", varargin)`
	`Y3 = varargout(0)`
	`Y4 = varargout(1)`
	`'omit some inputs and outputs`
	`Call com.foo(1, Y1, Null, Null , X1, Null , Null )`

The optional `varargin`/`varargout` parameters are used by functions with mutable input/output arguments. When present, the `varargin`/`varargout` parameters are always listed as the last input parameters and the last output parameters. Both parameters are `VARIANT` arrays, each element representing one additional input or output argument. The `nArgOut` parameter also counts arguments which are collected by `varargout`.

All parameters other than `nArgOut` are passed as COM `VARIANT` types. Data Conversion lists the rules for conversion between MATLAB arrays and COM `VARIANT`s.

3.2.2 .NET Component

All classes of the .NET API are organized in the namespace `PEAXACT`.

For each MATLAB function, the .NET component has overloaded methods to implement the various forms of a generic MATLAB function call.

- A **single output** signature that assumes that only a single output is required and returns the result in a single `MWArray`.
- A **standard** signature that specifies inputs of type `MWArray` and returns values as an array of `MWArray`.
- A **feval** signature that includes both input and output arguments in the argument list rather than returning outputs as a return value. Output arguments are specified first, followed by the input arguments. This interface is not documented here. It is recommended to use one of the other interfaces instead.

The following tables show the mapping of a generic MATLAB function signature to C#.

Single output interface

Typically you use the single output interface for MATLAB functions that return a single argument. You can also use the single output interface when you only require the first output of a function or when you want to use the first output as the input to another function.

The single output API for a MATLAB function returns a single `MWArray` value `Y1`.

Signature	Sample
MATLAB	`function [Y1, Y2, ...] = foo(X1, X2, ...)`
C#	`public MWArray foo(MWArray X1, MWArray X2, ...)`

The input arguments `X1`, `X2`, ... are `MWArray` types or supported .NET primitive types.

Note: In MATLAB, all inputs to functions are optional and may be present or omitted from the function call. The API provides several forms of the single output interface for different numbers of inputs.

Standard interface

Typically you use the standard interface for MATLAB functions that return multiple output values.

The standard calling interface returns an array of `MWArray` objects rather than a single array object.

Signature	Sample
MATLAB	`function [Y1, Y2, ...] = foo(X1, X2, ...)`
C#	`public MWArray[] foo(int numArgsOut, MWArray X1, MWArray X2, ...)`

The first argument `numArgsOut` is an integer. It is the number of requested output arguments. `numArgsOut` must be smaller or equal to the total number of possible output arguments. Outputs `Y1`, `Y2`, ... are returned as the elements of an array of `MWArrays`.

Mutable inputs and outputs

Some MATLAB functions specify an optional `varargin` and/or `varargout` parameter for mutable input/output arguments. When present, the `varargin`/`varargout` parameters are always listed as the last input parameters and the last output parameters. Both parameters are arrays, each element representing one additional input or output argument.

Signature	Sample
MATLAB	`function [Y1, Y2, ..., varargout] = foo(X1, X2, ..., varargin)`
C# (single output interface)	`public MWArray foo(MWArray X1, MWArray X2, ..., params MWArray[] varargin)`
C# (standard interface)	`public MWArray[] foo(int numArgsOut, MWArray X1, MWArray X2, ..., params MWArray[] varargin)`

When the `varargin` parameter is present in the MATLAB function, you can specify optional inputs: list the optional inputs, or put them in an `MWArray[]` argument, placing the array last in the argument list.

When the `varargout` parameter is present in the MATLAB function, you can use the standard calling interface to get all output arguments returned as an array of `MWArrays`, including those collected by `varargout`.

3.3 Data Conversion Rules

This section describes the data conversion rules. When a method is invoked on the PEAXACT COM component or .NET component, the input parameters are converted to MATLAB internal array format and passed to the compiled MATLAB function. When the function exits, the output parameters are converted back.
In MATLAB, a ***matrix*** is the basic building block for all of the common data types. These include empty matrices (matrices with at least one dimension equal to zero), scalars (1-by-1 matrices), vectors (matrices with only one row or column) and regular 2D matrices.

3.3.1 COM Component

The COM client passes all input and output arguments in the compiled MATLAB functions as type `VARIANT`. The COM `VARIANT` type is a union of several simple data types. A type `VARIANT` variable can store a variable of any of the simple types, as well as arrays of any of these values.

Conversion from COM to MATLAB

VARIANT Type	Corresponding C/C++ Type	Corresponding Visual Basic Type	MATLAB Data Type
`VT_EMPTY`	-	-	`empty double`
`VT_BOOL`	`VARIANT_BOOL`(1)	`Boolean`	`logical`
`VT_R8`	`Double`	`Double`	`double`
`VT_BSTR`	`BSTR`(1)	`String`	`char`
`VT_VARIANT|VT_ARRAY`	`VARIANT[]`(1)	`Variant()`	`cell`
Others			Not used

(1) Denotes Windows specific type. Not part of standard C/C++.

Conversion from MATLAB to COM

MATLAB Data Type	VARIANT Type for Empty Data	VARIANT Type for Scalar Data	VARIANT Type for Array Data
`logical`	`VT_EMPTY`	`VT_BOOL`	`VT_BOOL|VT_ARRAY`
`double`	`VT_EMPTY`	`VT_R8`	`VT_R8|VT_ARRAY`
`char`	`VT_EMPTY`	`VT_BSTR` with length = 1	`VT_BSTR` with a length > 1
`cell`	`VT_EMPTY`	`VARIANT` with a type conforming to the conversion rule for the MATLAB data type of the cell contents	`VARIANT` of type `VT_VARIANT|VT_ARRAY` with the type of each array member conforming to the conversion rule for the MATLAB data type of the corresponding cell

MATLAB Data Type	VARIANT Type for Empty Data	VARIANT Type for Scalar Data	VARIANT Type for Array Data
`struct`	`VT_EMPTY`	A MATLAB struct array is converted to an `MWStruct` object (see MATLAB Utility Library). This object is passed as a `VT_DISPATCH` type.	
Others	Not used	Not used	Not used

Note: The COM client returns vectors and matrices of higher dimension as `VT_ARRAY`. Indexing into these `VT_ARRAY`s is **one based** rather than zero based.
Note: Whenever a method of the Application Server returns arguments of variable dimensions, you should test for all cases: empty, scalar, or array.

3.3.2 .NET Component

To support data conversion between managed types and MATLAB types, the MATLAB Utility Library provides a set of data conversion classes derived from the abstract class, `MWArray`. When you invoke a method on a component, the input and output parameters are derived types of `MWArray`.

Conversion from .NET to MATLAB

.NET Type (Class)	.NET Type (Native)	MATLAB Data Type
`MWLogicalArray`	`System.Boolean`	`logical`
`MWNumericArray`	`System.Double`	`double`
`MWCharArray`	`System.String`	`char`
`MWCellArray`	-	`cell`
Others	Not used	Not used

To pass parameters, you can either instantiate one of the `MWArray` subclasses explicitly, or, for managed types `System.Double` and `System.String`, rely on implicit data conversion.

Note: `MWArray`s have 2 dimensions and indexing is **one based** rather than zero based. Implicit conversion of 1D native arrays results in `MWArray`s with the first dimension being 1 (row vector).

Conversion from MATLAB to .NET

MATLAB Data Type	.NET Type (Class)	.NET Type (Native)
`logical`	`MWLogicalArray`	`System.Boolean`
`double`	`MWNumericArray`	`System.Double`
`char`	`MWCharArray`	`System.String`
`cell`	`MWCellArray`	-

MATLAB Data Type	.NET Type (Class)	.NET Type (Native)
struct	MWStructArray	-
Others	Not used	Not used

All variables returned from MATLAB are represented as instances of the appropriate MWArray subclass. For example, a MATLAB cell array is returned as an MWCellArray object.

> Note: MWArrays have 2 dimensions and indexing is **one based** rather than zero based. In the special case of empty arrays at least one dimension is 0; for scalars both dimensions are 1.

- Use ToArray() method in order to convert an MWLogicalArray, MWNumericArray, or MWCharArray to a 2-dimensional native array.
- Use ToScalarInteger(), ToScalarDouble(), etc. in order to convert an MWNumericArray to a native scalar.
- Use ToString() in order to convert an MWArray to a 1-dimensional native string.
- Conversion of an MWCellArray requires element-wise conversion.
- Conversion of an MWStructArray requires field-wise conversion.

3.4 MATLAB Utility Library

3.4.1 COM Component

The MWComUtil type library includes helper classes for array processing and data conversion. In particular, the library provides two classes MWStruct and MWField for processing the MATLAB struct data type returned by some PEAXACT methods.

> Note: The MWComUtil type library is contained in the file mwcomutil.dll which is installed and registered during installation of the MATLAB Compiler Runtime (MCR). By default, the file is located at
> MCR_INSTALL_DIRECTORY\v714\runtime\win32|win64

MWStruct Class

The MWStruct class holds a MATLAB struct type. The struct is a container using named fields for storing other data types.

Property Item([fieldName]) As MWField

The Item property is the default property of the MWStruct class. This property is used to get/set a particular field in the structure.

Property NumberOfFields As Long

The read-only NumberOfFields property returns the number of fields in the structure.

Property FieldNames As Variant

The read-only `FieldNames` property returns an array of length `NumberOfFields` that contains the field names of the elements of the structure.

MWField Class

The `MWField` class holds a single field reference in an `MWStruct` object.

Property Name As String

The name of the field (read only).

Property Value As Variant

Stores the field's value (read/write). The `Value` property is the default property of the `MWField` class. The value of a field can be any type that is coercible to a `Variant`, as well as object types.

Example: Processing a MATLAB struct type in Visual Basic 6

Note: Before using `MWStruct` and `MWField` classes, you must make explicit reference to the MWComUtil 7.14 Type Library in your Microsoft Visual Basic IDE.

```
Sub foo ()
    Dim x As MWStruct
    Dim y As Variant
    Dim FieldName As Variant

    On Error Goto Handle_Error
    '
    '... Call a method that returns a scalar MWStruct in x
    '
    For Each FieldName In x.FieldNames
        y = x.Item(FieldName).Value ' or simply y = x(FieldName).Value
        ' ... Check whether y is nothing (empty), scalar, or an array
        ' ... Do something with y
    Next
    Exit Sub
Handle_Error:
    MsgBox(Err.Description)
End Sub
```

3.4.2 .NET Component

The `MWArray` assembly is a helper library providing data conversion classes. You reference this assembly and specify the namespace

`MathWorks.MATLAB.NET.Arrays`

in your managed application to convert native arrays to MATLAB arrays and vice versa. The assembly is located at

`INSTALLDIR\DLL\.NET\MWArray.dll`

MWArray Class

The data conversion classes are built as a class hierarchy that represents the major MATLAB array types. The root of the hierarchy is the `MWArray` abstract class. The `MWArray` class has the following subclasses representing the major MATLAB types:

- `MWNumericArray`
- `MWLogicalArray`
- `MWCharArray`
- `MWCellArray`, and
- `MWStructArray`

> Note: As in MATLAB, `MWArray`s have **2 dimensions** and indexing is **one based** rather than zero based.

`MWArray` and its derived classes provide the following functionality:

- Constructors and destructors to instantiate and dispose of MATLAB arrays
- Properties to get and set the array data
- Indexers to support a subset of MATLAB array indexing
- Implicit and explicit data conversion operators
- General methods

> Note: For complete reference information about the `MWArray` class hierarchy, see the `MWArray` Class Library Reference, available online only at http://www.mathworks.de/help/releases/R2010b/toolbox/dotnetbuilder/MWArrayAPI/HTML/index.html

3.5 Class Reference

This Section describes all methods of the PEAXACT Application Server API. Function signatures are noted in MATLAB syntax

```
function [out1, out2, ...] = functionName(in1, in2, ...)
```

These signatures can be translated to COM and .NET function calls according to the Calling Conventions and Data Conversion Rules.

3.5.1 Class Design

The API exposes two classes:

- Class Toolbox provides analysis methods of PEAXACT Toolbox
- Class Chrom provides analysis methods of PEAXACT Chrom

Both classes have some methods in common, e.g. for initialization.

Singleton Class Instance

The Application Server API implements a design pattern which enforces each application/process to use a single class instance only. Although it is possible to create multiple instances of the same class, internally all instances created by the same process will share the same workspace.

The benefit of this design is an efficient memory usage. Also, it enables you to initialize the license and logger only once per process.

In order to perform multiple parallel analyses, the Toolbox class provides session domains. Each session is an isolated environment where analyses execute. The Chrom class is a simplistic class which doesn't need sessions.

> Note: It is recommended to use a single instance of the Toolbox or Chrom class only. If you used multiple instances they would effectively behave like clones.
> Note: It is recommended NOT to use instances of both the Toolbox and Chrom class within the same process.
> Note: It is recommended NOT to use both the COM API and .NET API within the same process.

Exception Handling

The API is designed to not throw any exceptions. Instead, most methods return a Boolean variable `isOK` which is `false` if an error occurred while executing the method. In that case, `getLastErrorMessage()` can be called to get the corresponding error message. The calling function should be responsible for recovering from this state.

3.5.2 Common Methods

The following methods are members of both the Toolbox class and Chrom class.

initialize()

```
function [isOK] = initialize()
```

Initialize the Application Server.
Call this method in order to explicitly initialize the Application Server. Otherwise, this method is called implicitly when required. However, explicit initialization is recommended to determine whether it was successful. Initialization covers:

- Creation of a default file logger if none exists. Use `setLoger()` before or after `initialize()` to customize the logger.
- License validation as well as license activation. In case no valid license is found, the License Activation Dialog is shown for the user to load a license interactively. Use `setLicense()` before `initialize()` to set a license programmatically and prevent showing the license activation dialog.
- Creation of a new empty session.

`isOK (scalar logical)` is `false` in case of errors.

Note: Calling this method multiple times has no effect when initialization was successful before.

isInitialized()

```
function [isOK] = isInitialized()
```

Query status of initialization

`isOK (scalar logical)` is `true` (`false`) if the Application Server is (not) initialized.

getLastErrorMessage()

```
function [message] = getLastErrorMessage()
```

Get message of last caught exception.

`message (string)` is the text of the last caught exception. It typically changes when the `isOK` return value of other methods is `false`.

Note: Calling this method also resets the last error message to an empty string.

getLogger()

```
function [logLevel, logFilename] = getLogger()
```

Get logging details.

`logLevel (string)` is the current state of the logging level.

`logFilename (string)` is the current log filename.

If no logger is set, `logLevel` is "`OFF`" and `logFilename` is an empty string.

setLicense()

```
function [isOK] = setLicense()
```

Activate license.

If no license has been set so far (by either `initialize()` or any of the `setLicense()` methods) the Windows registry is searched for a permanently activated license.

`isOK (scalar logical)` is `true` if a valid license could be found and `false` if not.

```
function [isOK] = setLicense(filename)
```

Activate specific license.

This method gives you full control over the license to be used. It should be called before `initialize()`. However, it can be called anytime to switch the license.

`filename (string)` is the fully qualified filename of a PEAXACT license.

`isOK (scalar logical)` is `false` if the specified license is invalid.

```
function [isOK] = setLicense(filename, OEM_Key)
```

Activate OEM license.

OEM licenses are available for third-party software developers only who integrate PEAXACT with their software and re-sell it as part of their own products to end-customers.

`filename (string)` is the fully qualified filename of a PEAXACT OEM license.
`OEM_Key (string)` is a key code which activates the OEM license.
`isOK (scalar logical)` is `false` if the specified license is invalid.

setLogger()

`function [isOK] = setLogger()`

Create default file logger if none exists.
If no file logger exists, a new one is created with log level `INFO`. Otherwise, this method does nothing.
`isOK (scalar logical)` is `false` if no logger could be created which means that logging is disabled.

`function [isOK] = setLogger(logLevel)`

Create new file logger or change existing logger.
If no file logger exists, a new one is created with log level `logLevel`. If a file logger does exist, its log level is updated to `logLevel`.
`logLevel (string)` can be one of the following strings: ALL or DEBUG, EXCEPTION, INFO, WARNING, SEVERE, OFF.
`isOK (scalar logical)` is `false` if no logger could be created or if `logLevel` is invalid.

`function [isOK] = setLogger(logLevel, logFilename)`

Create new logger or change existing logger.
If no file logger exists, a new one is created with log level `logLevel` and log file `logFilename`. If a file logger does exist, its log level is updated and its file is moved to `logFilename`.
`logLevel` (see above)
`logFilename (string)` is the full path and filename of the log file.
`isOK (scalar logical)` is `false` if no logger could be created or if the log file could not be moved.

Note: If this method is not called explicitly, a file logger is created implicitly with `logLevel = INFO` and a default `logFilename`.
Note: If no file logger exists when this method is called (explicitly or implicitly) but the log file could not be opened, `logLevel` will be set to `OFF`.

terminate()

`function [isOK] = terminate()`

Uninitialize the Application Server.
Calling this method explicitly terminates the Application Server and returns whether termination was successful. If this method is not called explicitly, termination is done implicitly when required. Termination covers:

- Removal of the file logger (if any), including closing of the log file
- Deactivation of the license
- Removal of all sessions and runtime data

`isOK (scalar logical)` is `true` (`false`) if termination was (not) successful.

Note: It is not recommended to call any other method after `terminate()` because this may implicitly re-initialize the interface.

testFunctionIO()

`function [isOK, outputStruct, varargout] = testFunctionIO(x1, x2, varargin)`

This function provides a testing platform for the calling conventions and data conversion rules of MATLAB functions. For any input argument you pass, the function displays a message box with details about the argument's type, size, and value from the MATLAB point of view. Output arguments have predefined values. In particular, `outputStruct` is a MATLAB struct with fields of all kinds of matrix dimensions and data types.

`x1 (any size and type)` optional first input, default value = "default x1"

`x2 (any size and type)` optional second input, default value = "default x2"

`varargin (any size, cell)` optional further inputs, collected in a cell array

`isOK (scalar logical)`, constant = `true`

`outputStruct (scalar struct)` is a struct with fields holding predefined values of all kinds of matrix dimensions and data types. The name of each field also describes the data type of the field's value. The following fields exist:

- `emptyLogical`, `emptyDouble`, `emptyChar`, `emptyCell`, `emptyStruct`
- `scalarLogical`, `scalarDouble`, `scalarChar`, `scalarCellOfScalarLogical`, `scalarCellOfScalarDouble`, `scalarCellOfScalarChar`
- `vectorLogical`, `vectorDouble`, `vectorChar`, `vectorCellOfScalarLogical`, `vectorCellOfScalarDouble`, `vectorCellOfScalarChar`, `vectorCellOfVectorLogical`, `vectorCellOfVectorDouble`, `vectorCellOfVectorChar`, `vectorCellOfVectorCellOfStrings`
- `matrixLogical`, `matrixDouble`, `matrixChar`, `matrixCellOfScalarDouble`

`varargout (size of varargin cell)` is a copy of `varargin`.

3.5.3 Toolbox Class Methods

activateSession()

`function [isOK, activeSessionID] = activateSession()`

`function [isOK, activeSessionID] = activateSession(sessionID)`

Make session active.

Methods of class `Toolbox` implicitly operate on the active session. If you have multiple sessions you can use this method to switch between them.

`sessionID (string)` is the unique identifier of a session to be activated. If missing or empty, the active session is not changed.

`isOK (scalar logical)` is `false` in case of errors.

`activeSessionID (string)` is the unique identifier of the active session (after switching) or empty if `isOK` is `false`.

Note: If you plan to use parallel sessions, call `activateSession()` once without inputs after `initialize()` in order to retrieve the session ID of the initial session.

newSession()

```
function [isOK, activeSessionID] = newSession()
```

Add a new session and make it the active one.
A session is a container for models and data sets. By default, the `Toolbox` instance already has one initial session. Use this method to create parallel sessions.
`isOK (scalar logical)` is `false` in case of errors.
`activeSessionID (string)` is the unique identifier of the new session or empty if `isOK` is `false`. You should keep track of session IDs in order to switch between sessions with `activateSession()` or remove sessions with `removeSession()`.

removeSession()

```
function [isOK, activeSessionID] = removeSession(sessionID)
```

Remove a session.
`sessionID (string)` is the unique identifier of a session to be removed. If it is the active session, the next session will be activated. If there are no further sessions, a new session will be added such that always at least one session exists.
`isOK (scalar logical)` is `false` in case of errors.
`activeSessionID (string)` is the unique identifier of the active session (after removal) or empty if `isOK` is `false`.

addDataSet()

```
function [isOK] = addDataSet(URI)
function [isOK] = addDataSet(URI, varargin)
```

Add data set with optional additional features.
`URI (string)` is the full path, filename, and access ID of a sample in a data file. You cannot add the same `URI` more than once. Trying so will simply update the existing data set.
Alternative 1: `varargin = missing or empty`
If `varargin` is omitted from the function call or empty, the data set is added without any additional features. xy-values will be read from the data file.
Alternative 2: `varargin = {xData, yData}`
Alternative 3: `varargin = {..., key1, value1, key2, value2, ...}`
Alternative 4: `varargin = {..., "clear"}`
`xData` and `yData (*-by-1 double)` are optional column vectors of x-values and y-values to be used instead of reading xy-values from a data file. In this case `URI` can be a dummy filename. If provided, `xData` and `yData` must be the first two elements of `varargin`.
`key (string)` and `value (scalar double)` are the name and value of an additional feature added to the data set. You can optionally provide any number of key/value pairs.

`"clear" (string literal)` is an optional switch. If provided, all previously added data sets (except any which matches `URI`) will be removed before adding the new data set. Option `clear` must be the last element of `varargin`.

`isOK (scalar logical)` is `false` in case of errors.

addModel()

```
function [isOK] = addModel(fullFilename)
```

Add master model from file.

`fullFilename (string)` is the full path and filename of

- either a PEAXACT model file,
- or a PEAXACT session file containing one or more models

created and saved with PEAXACT Toolbox. In case of a model file, a single model is added. In case of a session file, all models contained in the session will be added. Note that models taken from a session will get a new unique filename which is composed of the session filename and the model name. This filename is just used as an identifier (models do not have to be present as model files on the hard disk). The unique identifier enables you to load models with identical names from multiple sessions and still be able to distinguish between them. Use `getModelFilenames()` to get a list of all added model filenames.

`isOK (scalar logical)` is `false` in case of errors.

analysisPeakSearch()

```
function [isOK, outputStruct] = analysisPeakSearch()
function [isOK, outputStruct] = analysisPeakSearch(minPeakHeight)
```

Perform Peak Search of all added data sets using data pretreatment settings of the first added master model (if any).

Alternative 1: `minPeakHeight = missing or empty`

If `minPeakHeight` is omitted from the function call or empty, the minimum peak detection height is estimated automatically and individually for each data set.

Alternative 2: `minPeakHeight (scalar double)` is the minimum height for peaks to get detected, while peaks smaller than `minPeakHeight` are ignored.

`isOK (scalar logical)` is `false` in case of errors.

`outputStruct (scalar struct)` is a result struct with fields:

- `nDataSets (scalar double)` is the number of processed data sets.
- `dataSetURIs (nDataSets-by-1 cell of strings)` is a cell array of URIs of all processed data sets.
- `minPeakHeight (nDataSets-by-1 double)` is the matrix of actually used peak detection heights. Values are equal to input argument `minPeakHeight` if provided.
- `peakPositionIndices (nDataSets-by-1 cell of *-by-1 double)` is a cell array where each element is a column vector of indices of `xData` at which peaks are found.

- `peakPositions (nDataSets-by-1 cell of *-by-1 double)` is a cell array where each element is a column vector of x-values at which peaks are found, i.e. `peakPositions = xData(peakPositionIndices)`.
- `peakIntensities (nDataSets-by-1 cell of *-by-1 double)` is a cell array where each element is a column vector of y-values at the peak positions, i.e. `peakIntensities = yData(peakPositionIndices)`.
- `xData (nDataSets-by-1 cell of *-by-1 double)` is a cell array where each element is a column vector of a sample's x-values.
- `yData (nDataSets-by-1 cell of *-by-1 double)` is a cell array where each element is a column vector of a sample's y-values.

analysisMCR()

```
function [isOK, outputStruct] = analysisMCR(nComponents)
function [isOK, outputStruct] = analysisMCR(nComponents, varargin)
```

Perform Multivariate Curve Resolution of all added data sets using data pretreatment settings of the first added master model (if any).

`nComponents (scalar double)` is the number of unknown components which should be identified from the samples' y-data.

Alternative 1: `varargin = missing or empty`

If `varargin` is omitted from the function call or empty, MCR is performed with default settings. This also re-uses previous results for initialization of concentrations (`C0`) when called repeatedly.

Alternative 2: `varargin = {key1, value1, key2, value2, ...}`

`key (string)` and `value (class and size differs)` are the name and value of optional settings for MCR. The following keys are recognized:

- `C0 (empty or nDataSets-by-nComponents double)` is an optional matrix of initial concentrations. If `C0` is empty (`Null`), concentrations are initialized implicitly (reset). This is different from omitting the argument which would initialize `C0` with previous results (if available).
- `toleranceRMSE (1-by-1 double)` is an optional criterion for stopping MCR when progress of iterations drops below the tolerance (`default = 1e-5`).
- `nIterations (1-by-1 double)` is an optional criterion for stopping MCR after a maximum number of iterations is reached (`default = 100`).
- `nUnsuccessfulAttempts (1-by-1 double)` is an optional criterion for stopping MCR after a maximum number of unsuccessful iterations (`default = 20`).
- `isNonnegativeC (1-by-1 logical)` is an optional switch to enable or disable the non-negativity constraint of concentrations (`default = false`).
- `isNonnegativeS (1-by-1 logical)` is an optional switch to enable or disable the non-negativity constraint of component spectra (`default = false`).
- `isUnimodalC (1-by-1 logical)` is an optional switch to enable or disable the unimodality constraint of component concentrations (`default = false`).
- `isClosureC (1-by-1 logical)` is an optional switch to enable or disable the closure constraint of component concentrations (`default = false`).

`isOK (scalar logical)` is `false` in case of errors.

`outputStruct (scalar struct)` is a result struct with fields:

- `nDataSets (scalar double)` is the number of processed data sets.
- `dataSetURIs (nDataSets-by-1 cell of strings)` is a cell array of URIs of all processed data sets.
- `nComponents (scalar double)` is the number of components being identified from the data. This simply is a copy of the input argument.
- `componentNames (1-by-nComponents cell of strings)` is a cell array of automatically generated names for each identified component.
- `S (*-by-nComponents double)` is a matrix of component spectra identified from the samples' y-data. The matrix has as many rows as `xData`.
- `C (nDataSets-by-nComponents double)` is a matrix of component concentrations for each data set identified from the samples' y-data.
- `RMSResiduals (nDataSets-by-1 double)` is a vector of root mean square residuals for each sample. Residuals are calculated by $yData - SC^T$.
- `R2 (scalar double)` is the fraction of variance in `yData` explained by SC^T.
- `xData (*-by-1 double)` is column vector of x-values at which `S` is calculated.

analysisHMFA()

```
function [isOK, outputStruct] = analysisHMFA(nUnknownComponents)
function [isOK, outputStruct] = analysisHMFA(nUnknownComponents, varargin)
```

Perform Hard Modeling Factor Analysis of all added data sets using the first added master model which contains a hard model.

`nUnknownComponents (scalar double)` is the number of unknown components which should be identified from the samples' y-data.

Alternative 1: `varargin = missing or empty`

If `varargin` is omitted from the function call or empty, HMFA is performed with default settings.

Alternative 2: `varargin = {key1, value1, key2, value2, ...}`

`key (string)` and `value (class and size differs)` are the name and value of optional settings for HMFA. The following keys are recognized:

- `isClosureC (1-by-1 logical)` is an optional switch to enable or disable the closure constraint of component concentrations (`default = false`).

`isOK (scalar logical)` is `false` in case of errors.

`outputStruct (scalar struct)` is a result struct with fields:

- `nDataSets (scalar double)` is the number of processed data sets.
- `dataSetURIs (nDataSets-by-1 cell of strings)` is a cell array of URIs of all processed data sets.
- `nComponents (scalar double)` is the sum of known components from the hard model and identified unknown components (`nUnknownComponents`) from the data.
- `componentNames (1-by-nComponents cell of strings)` is a cell array of component names taken from the hard model (if any) as well as automatically generated names for each identified component (if any).

- `S (*-by-nComponents double)` is a matrix of component spectra identified from the samples' y-data. The matrix has as many rows as `xData`.
- `C (nDataSets-by-nComponents double)` is a matrix of component concentrations for each data set identified from the samples' y-data.
- `RMSResiduals (nDataSets-by-1 double)` is a vector of root mean square residuals for each sample. Residuals are calculated by `yData - SC`^{T}.
- `R2 (scalar double)` is the fraction of variance in `yData` explained by `SC`^{T}.
- `xData (*-by-1 double)` is column vector of x-values at which `S` is calculated.

analysisComponentFitting()

```
function [isOK, outputStruct] = analysisComponentFitting()
function [isOK, outputStruct] = analysisComponentFitting(componentIdentifier)
```

Perform Component Fitting of all added data sets using all added models which contain a hard model.

Alternative 1: `componentIdentifier = missing or empty`

If `componentIdentifier` is omitted from the function call or empty, results are returned for all hard model components contained in all added master models.

Alternative 2: `componentIdentifier = componentNames`

`componentNames (1-by-* cell of strings)` is a cell array of hard model component names for which results should be returned. Use `getComponentNames()` to get a list of available names. Caution: If `componentNames` contains names that exist in more than one hard model, results are only returned for the last matching component. Use Alternative 3 in this case.

Alternative 3: `componentIdentifier = componentIndices`

`componentIndices (1-by-* double)` is a vector of hard model component indices for which results are returned. Indices correspond to names returned by `getComponentNames()`.

`isOK (scalar logical)` is `false` in case of errors.

`outputStruct (scalar struct)` is a result struct with fields:

- `nDataSets (scalar double)` is the number of processed data sets.
- `dataSetURIs (nDataSets-by-1 cell of strings)` is a cell array of URIs of all processed data sets.
- `nComponents (scalar double)` is the number of components for which weights are returned.
- `componentNames (1-by-nComponents cell of strings)` is a cell array of hard model component names for which weights are returned.
- `componentIndices (1-by-nComponents double)` is a vector of hard model component indices for which weights are returned.
- `w (nDataSets-by-nComponents double)` is a matrix of component weights for all processed data sets and for all requested hard model components.

analysisIntegration()

```
function [isOK, outputStruct] = analysisIntegration()
function [isOK, outputStruct] = analysisIntegration(integrationRangeIdentifier)
```

Perform Integration of all added data sets using all added models which contain an integration model.

Alternative 1: `integrationRangeIdentifier = missing or empty`

If `integrationRangeIdentifier` is omitted from the function call or empty, results are returned for all integration ranges contained in all added master models.

Alternative 2: `integrationRangeIdentifier = integrationRangeNames`

`integrationRangeNames (1-by-* cell of strings)` is a cell array of integration range names for which results should be returned. Use `getIntegrationRangeNames()` to get a list of available names. Caution: If `integrationRangeNames` contains names that exist in more than one integration model, results are only returned for the last matching integration range. Use Alternative 3 in this case.

Alternative 3: `integrationRangeIdentifier = integrationRangeIndices`

`integrationRangeIndices (1-by-* double)` is a vector of integration range indices for which results are returned. Indices correspond to names returned by `getIntegrationRangeNames()`.

`isOK (scalar logical)` is `false` in case of errors.

`outputStruct (scalar struct)` is a result struct with fields:

- `nDataSets (scalar double)` is the number of processed data sets.
- `dataSetURIs (nDataSets-by-1 cell of strings)` is a cell array of URIs of all processed data sets.
- `nIntegrationRanges (scalar double)` is the number of integration ranges for which peak areas are returned.
- `integrationRangeNames (1-by-nIntegrationRanges cell of strings)` is a cell array of integration range names for which peak areas are returned.
- `integrationRangeIndices (1-by-nIntegrationRanges double)` is a vector of integration range indices for which peak areas are returned.
- `A (nDataSets-by-nIntegrationRanges double)` is a matrix of calculated peak areas for all processed data sets and for all requested integration ranges.

analysisPrediction()

```
function [isOK, outputStruct] = analysisPrediction()
function [isOK, outputStruct] = analysisPrediction(featureIdentifier)
```

Perform Prediction of all added data sets using all added models which contain a calibration model.

Alternative 1: `featureIdentifier = missing or empty`

If `featureIdentifier` is omitted from the function call or empty, results are returned for all calibrated features contained in all added master models.

Alternative 2: `featureIdentifier = featureNames`

`featureNames (1-by-* cell of strings)` is a cell array of calibrated feature names for which results should be returned. Use `getCalibratedFeatureNames()` to get a

list of available names. Caution: If `featureNames` contains names that exist in more than one calibration model, results are only returned for the last matching feature. Use Alternative 3 in this case.

Alternative 3: `featureIdentifier = featureIndices`

`featureIndices (1-by-* double)` is a vector of feature indices for which results should be returned. Indices correspond to names returned by `getCalibratedFeatureNames()`.

`isOK (scalar logical)` is `false` in case of errors.

`outputStruct (scalar struct)` is a result struct with fields:

- `nDataSets (scalar double)` is the number of processed data sets.
- `dataSetURIs (nDataSets-by-1 cell of strings)` is a cell array of URIs of all processed data sets.
- `nFeatures (scalar double)` is the number of calibrated features for which predicted values are returned.
- `featureNames (1-by-nFeatures cell of strings)` is a cell array of feature names for which predicted values are returned.
- `featureIndices (1-by-nFeatures double)` is a vector of feature indices for which predicted values are returned.
- `x (nDataSets-by-nFeatures double)` is a matrix of predicted values for all processed data sets and for all requested features.
- `RMSResiduals (nDataSets-by-nFeatures double)` is a matrix of root mean square (RMS) spectral residuals. The matrix has identical columns for features predicted by the same IHM model. The matrix contains NaN elements for features predicted by Peak Integration models.
- `RMSResidualsOutlierPValue (nDataSets-by-nFeatures double)` is a matrix of probability values (p-values) for each spectral residuals being an outlier. The matrix has identical columns for features predicted by the same IHM model. The matrix contains NaN elements for features predicted by Peak Integration models.
- `mahalanobisDistance (nDataSets-by-nFeatures double)` is a matrix of mahalanobis distances. The matrix contains NaN elements for features not predicted by PLS models.
- `mahalanobisDistanceOutlierPValue (nDataSets-by-nFeatures double)` is a matrix of probability values (p-values) for each distance being an outlier. The matrix contains NaN elements for features not predicted by PLS models.

getCalibratedFeatureNames()

```
function [featureNames] = getCalibratedFeatureNames()
```

Get names of calibrated features from all added calibration models.

`featureNames (1-by-* cell of strings)` is a cell array of available calibrated feature names, ordered as they appear in calibration models of added master models. The cell array is empty if no calibration models can be found or in case of errors.

Any combination of names in `featureNames` can be passed as input to `analysisPrediction()`. However, it is recommended to get indices of these names from the order of `featureNames` and then pass indices to `analysisPrediction()` because this is unambiguous in case of identical names.

getComponentNames()

```
function [componentNames] = getComponentNames()
```

Get names of non-empty component models from all added hard models.

`componentNames (1-by-* cell of strings)` is a cell array of available component model names, ordered as they appear in hard models of added master models. Note: Component models without peaks are ignored! The cell array is empty if no hard models can be found or in case of errors.

Any combination of names in `componentNames` can be passed as input to `analysisComponentFitting()`. However, it is recommended to get indices of these names from the order of `componentNames` and then pass indices to `analysisComponentFitting()` because this is unambiguous in case of identical names.

getIntegrationRangeNames()

```
function [integrationRangeNames] = getIntegrationRangeNames()
```

Get names of integration ranges from all added integration models.

`integrationRangeNames (1-by-* cell of strings)` is a cell array of available component model names, ordered as they appear in integration models of added master models. The cell array is empty if no integration models can be found or in case of errors.

Any combination of names in `integrationRangeNames` can be passed as input to `analysisIntegration()`. However, it is recommended to get indices of these names from the order of `integrationRangeNames` and then pass indices to `analysisIntegration()` because this is unambiguous in case of identical names.

getDataSetURIs()

```
function [URIs] = getDataSetURIs()
```

Get `URIs` of added data sets.

`URIs (*-by-1 cell of strings)` is a cell array of URIs of all added data sets (see `addDataSet()`). The cell array is empty if no data sets are added or in case of errors.

getModelFilenames()

```
function [fileNames] = getModelFilenames()
```

Get `fileNames` of added master models.

`fileNames (1-by-* cell of strings)` is a cell array of filenames of all added master models (see `addModel()`). The cell array is empty if no models are added or in case of errors.

getModelInfo()

```
function [infoStruct] = getModelInfo(modelIndex)
function [infoStruct] = getModelInfo(modelFilename)
```

Get information about added master model

`modelIndex (scalar double)` is the index of a currently added master model. The index must be in between `1` and `nModels`.

`modelFilename (string)` is the full path and name of an added master model. The filename must match any name returned by `getModelFilenames()`.

`infoStruct (scalar struct)` is a struct with fields:

- `filename (string)` is the filename of the added model.
- `description (string)` is a free text provided by the creator of the model. The text is intended to describe the model but it may contain any kind of information. The text may contain line breaks.
- `nIntegrationRanges (scalar double)` is the number of integration ranges contained in the model.
- `nComponents (scalar double)` is the number of non-empty hard model components contained in the model.
- `nCalibratedFeatures (scalar double)` is the number of calibrated features contained in the model.
- `integrationRangeNames (1-by-* cell of strings)` is a cell array as returned by `getIntegrationRangeNames()`.
- `componentNames (1-by-* cell of strings)` is a cell array as returned by `getComponentNames()`.
- `featureNames (1-by-* cell of strings)` is a cell array as returned by `getCalibratedFeatureNames()`.

nDataSets()

```
function [N] = nDataSets()
```

Get number of added data sets.

`N (scalar double)` is the number of currently added data sets.

nModels()

```
function [M] = nModels()
```

Get number of added models.

`M (scalar double)` is the number of currently added master models.

removeDataSet()

```
function [isOK] = removeDataSet(dataSetIndex)
```

`function [isOK] = removeDataSet(dataSetURI)`

Remove data set.

`dataSetIndex (scalar double)` is the index of a currently added data set to be removed. The index must be in between `1` and `nDataSets`.

`dataSetURI (string)` is the URI of an added data set. The URI must match any name returned by `getDataSetURIs ()`.

`isOK (scalar logical)` is `false` in case of errors.

removeModel()

`function [isOK] = removeModel(modelIndex)`

`function [isOK] = removeModel(modelFilename)`

Remove model

`modelIndex (scalar double)` is the index of a currently added master model to be removed. The index must be in between `1` and `nModels`.

`modelFilename (string)` is the full path and name of an added master model. The filename must match any name returned by `getModelFilenames()`.

`isOK (scalar logical)` is `false` in case of errors.

3.5.4 Chrom Class Methods

analysisClassification()

`function [isOK, outputStruct] = analysisClassification(modelFile, dataSetURIs)`

Perform Classification of chromatograms.

`modelFile (string)` is the full path and filename of a PEAXACT model created and saved with PEAXACT Chrom Builder.

`dataSetURIs (string, or cell of strings)` is the full path and filename of a data file. It can also be a cell array of multiple filenames.

`isOK (scalar logical)` is `false` in case of errors.

`outputStruct (scalar struct)` is a result struct with fields:

- `nDataSets (scalar double)` is the number of processed data sets.
- `dataSetURIs (nDataSets-by-1 cell of strings)` is a cell array of URIs of all processed data sets. This could differ from the input argument `dataSetURIs` if some data sets haven't been processed.
- `classificationCode (nDataSets-by-1 double)` is a column vector with classification codes for each processed data set.
- `classificationText (nDataSets-by-1 cell of strings)` is a cell array where each element contains a string describing the result of classification.
- `yDataBaselineCorrected (nDataSets-by-1 cell of *-by-1 double)` is a cell array where each element contains sample y-values after baseline correction.
- `componentRetentionTimeShifts (nDataSets-by-1 cell of *-by-1 double)` is a cell array where each element contains retention time shifts for all model components marked as "significant".

- `missingComponentNames (nDataSets-by-1 cell of *-by-1 cell of strings)` is a cell array where each element again contains a cell array of model component names which are missing in the reference sample.
- `missingComponentRetentionTimes (nDataSets-by-1 cell of *-by-1 double)` is a cell array where each element contains retention times at which a model component is missing. Elements in the vector of retention times correspond to elements in the cell array of `missingComponentNames`.
- `additionalComponentRetentionTimes (nDataSets-by-1 cell of *-by-1 double)` is a cell array where each element contains retention times at which an additional component is found.

3.6 Programming Examples

3.6.1 Using the .NET API in C#

This example demonstrates how to use the .NET API in C#. The program uses a calibrated model to predict features from a measured sample.

> Note: In Visual Studio, you have to reference the assemblies `PEAXACT` and `MWArray` first. See Before You Start

```
using System;
using MathWorks.MATLAB.NET.Arrays;

namespace PEAXACT
{
    class PredictionExample
    {
        PEAXACT.Toolbox pxToolbox;

        static void Main(string[] args)
        {
            PredictionExample example = new PredictionExample();
            example.run();
            Console.WriteLine("Press any key to continue.");
            Console.ReadKey();
        }

        private PredictionExample()
        {
            // initialize Application Server
            pxToolbox = new PEAXACT.Toolbox();
            if (!(MWLogicalArray)pxToolbox.setLogger("DEBUG", "d:\\peaxact.log"))
                throwLastError();
            if (!(MWLogicalArray)pxToolbox.setLicense("c:\\peaxact\\license.lic"))
                throwLastError();
            if (!(MWLogicalArray)pxToolbox.initialize())
                throwLastError();
        }

        private void run()
        {
            loadModel("c:\\peaxact\\model.pxm");
            loadData("c:\\peaxact\\sample.xyz");
            predictFeatures();
```

```
        }

        private void loadModel(String modelFilename)
        {
            if (!(MWLogicalArray)pxToolbox.addModel(modelFilename))
                throwLastError();
        }

        private void loadData(String dataSetURI)
        {
            // This function demonstrates how to add data sets with known xy-data
            // Alternatively, you could pass a filename.

            Int32 nx = 800; // number of data points
            Double[] xData; // vector of x-data
            Double[] yData; // vector of y-data

            // populate xData and yData
            xData = new Double[nx];
            yData = new Double[nx];
            // ...

            // addDataSet() expects xData and yData to be column vectors. Because a
            // native 1D Double array would implicitly be casted to a row vector,
            // you should use one of the many MWNumericArray() constructors
            MWNumericArray xColumn = new MWNumericArray(nx, 1, xData);
            MWNumericArray yColumn = new MWNumericArray(nx, 1, yData);
            if (!(MWLogicalArray)pxToolbox.addDataSet(dataSetURI, x, y))
                throwLastError();
        }

        private void predictFeatures()
        {
            // This function demonstrates how to deal with multiple return values
            // and how to work with the MWStructArray

            MWArray[] varargout = pxToolbox.analysisPrediction(2); // 2 outputs
            Boolean isOK = (MWLogicalArray)varargout[0];
            if (!isOK) throwLastError();
            MWStructArray outputStruct = (MWStructArray)varargout[1];

            // convert some fields of the result struct to native types
            // conversion of numerical scalars using the ToScalarXXX() methods
            Int32 nDataSets =
((MWNumericArray)outputStruct.GetField("nDataSets")).ToScalarInteger();
            Int32 nFeatures =
((MWNumericArray)outputStruct.GetField("nFeatures")).ToScalarInteger();

            // conversion of double matrices using the ToArray method
            Double[,] values = (Double[,])outputStruct.GetField("x").ToArray();

            // you can also work with the MWArray type directly,
            // but keep in mind that indexing into MWArrays is 1-based
            MWCellArray mwDataSetURIs =
(MWCellArray)outputStruct.GetField("dataSetURIs");

            // conversion of cell arrays requires element-wise conversion
            MWCellArray mwFeatureNames =
(MWCellArray)outputStruct.GetField("featureNames");
            String[] names = new String[nFeatures];
            for (int i = 0; i < nFeatures; i++)
                names[i] = mwFeatureNames[1, i + 1].ToString(); // 1-based indexing

            // display results
            for (int i = 0; i < nDataSets; i++)
```

```
            {
                Console.WriteLine(mwDataSetURIs[1, i + 1]); // 1-based indexing
                for (int j = 0; j < nFeatures; j++)
                    Console.WriteLine("{0} = {1}", names[j], values[i, j]);
            }
        }

        private void throwLastError()
        {
            throw new Exception(pxToolbox.getLastErrorMessage().ToString());
        }
    }
}
```

3.6.2 Using the COM API in VB.NET

This example demonstrates how to use the COM API in Visual Basic .NET by means of a `Wrapper` class, which simply wraps COM function calls into VB code.

> Note: In Visual Studio, you have to reference the COM DLL first. Also, you have to set the reference's property `Embeded Interop Type` to `False`.

```
Public Class Wrapper

#Region "Properties"
    Private pxToolbox As PEAXACT.Toolbox

    Public ReadOnly Property lastErrorMessage As String
        Get
            lastErrorMessage = ""
            pxToolbox.getLastErrorMessage(1, lastErrorMessage)
        End Get
    End Property
#End Region


#Region "Methods"
    ' Initialize() demonstrates how to initialize PEAXACT
    Sub Initialize(ByVal logLevel As String, ByVal logFilename As String)
        If pxToolbox Is Nothing Then
            pxToolbox = CreateObject("PEAXACT.Toolbox")
            pxToolbox.setLogger(0, DBNull.Value, logLevel, logFilename)
            Dim isOK = False
            pxToolbox.initialize(1, isOK)
            If Not isOK Then Throw New Exception("Failed to initialize PEAXACT." _
                                                 & Chr(13) & lastErrorMessage)
        End If
    End Sub

    ' RequireInitialization() assures that PEAXACT is initialized
    Private Sub RequireInitialization()
        Initialize("", "")
    End Sub

    ' AddModel() demonstrates how to add a model file
    Public Function AddModel(ByVal modelFilename As String) As Boolean
        RequireInitialization()
        AddModel = False
        pxToolbox.addModel(1, AddModel, modelFilename)
    End Function
```

```
    ' GetCalibratedFeatureNames() demonstrates how convert
    ' MATLAB cell array of strings into a native string array
    Public Function GetCalibratedFeatureNames() As String(,)
        RequireInitialization()
        Dim featureNames As Object(,) = {}
        GetCalibratedFeatureNames = {}

        pxToolbox.getCalibratedFeatureNames(1, featureNames)
        Dim featureNamesString(UBound(featureNames, 1) - 1, _
                               UBound(featureNames, 2) - 1) As String
        For iRow As Integer = 1 To UBound(featureNames, 1)
            For iCol As Integer = 1 To UBound(featureNames, 2)
                featureNamesString(iRow - 1, iCol - 1) = _
                                        CStr(featureNames(iRow, iCol))
            Next
        Next
        GetCalibratedFeatureNames = featureNamesString
    End Function

    ' Predict() demonstrates how to call PEAXACT functions with
    ' optional input arguments and how to handle type MWStruct variables
    Function Predict(ByVal dataSetURI As String) As Double(,)
        RequireInitialization()
        Dim isOK As Boolean
        Dim outputStruct As New MWComUtil.MWStruct
        Predict = {}

        ' add data set
        pxToolbox.addDataSet(1, isOK, dataSetURI, DBNull.Value)
        Predict = {{-1}}
        If Not CBool(isOK) Then Exit Function

        ' predict features
        pxToolbox.analysisPrediction(2, isOK, outputStruct, DBNull.Value)
        Predict = {{-2}}
        If Not CBool(isOK) Then Exit Function

        ' convert results
        Predict = CType(outputStruct("x").Value, Double(,))
    End Function
#End Region
End Class
```

3.6.3 Using the COM API in VB Script

This example demonstrates how to call the PEAXACT Application Server from a Visual Basic Script (VBS). VBS only supports the COM API. The script uses a model to classify two chromatograms and displays results. This example relies on implicit initialization of the Application Server, because `initialize()` is not called explicitly.

> Note: If you installed the 32 bit version of the Application Server on a 64 bit OS, you need to run the script with a 32 bit version of `wscript.exe`, which can be found at `%windir%\SysWOW64\wscript.exe`

```
' create instance of class PEAXACT.Chrom
Set pxChrom = CreateObject("PEAXACT.Chrom")

' create instance of utility class MWComUtil.MWStruct
Set outputStruct = CreateObject("MWComUtil.MWStruct7.14")
```

```
' classify chromatograms according to model specifications
modelFile = "c:\model\file.pxm"
dataSetURIs(0) = "c:\data\chromatogram.csv#1"
dataSetURIs(1) = "c:\data\chromatogram.csv#2"
pxChrom.analysisClassification 2, isOK, outputStruct, modelFile, dataSetURIs

' display results
' Note: Indexing into VARIANT arrays returned by the Application Server is 1-based
nDataSets = outputStruct.Item(1, "nDataSets") ' = 2
URIs = outputStruct.Item(1, "dataSetURIs") ' = dataSetURIs
code = outputStruct.Item(1, "classificationCode")
text = outputStruct.Item(1, "classificationText")
For i  = 1 To nDataSets
    Wscript.Echo "Classification of " & URIs(i) _
        & ": Code " & code(i) _
        & ", Description: " & text(i)
Next
```

4 CUSTOM INTERFACES

4.1 OPUS Process

Note: A video on how to integrate the PEAXACT Application Server with OPUS Process is published on the PEAXACT Blog.

4.1.1 Prerequisites

Software Requirements

- OPUS 6.5 or higher
- PEAXACT 3.6.0 or higher. A 32-bit installation is required!

OPUS 7 Workaround

The following workaround is necessary for OPUS version 7 to work with PEAXACT:

1) Open the Windows Explorer and open the OPUS installation directory
2) Rename file `Calo.dll` to `Calo.dll_hidden` or any other name, such that the file won't be found be OPUS any more

Please note that this workaround disables OPUS support for Unscrambler.

Additional Files

These instructions refer to a special OPUS script file named `PEAXACTComponentAnalysis.obs`. The file is used as a placeholder during set-up of an OPUS PROCESS scenario and does nothing so far. The file is located at `INSTALLDIR\DLL\OPUS`.

4.1.2 OPUS Configuration

1) Run the diagnosis program first to test whether the PEAXACT Application Server (COM API) is installed and registered correctly.
2) Configure a new OPUS PROCESS scenario file (.obs) with the OPUS scenario browser
   a) Each measurement point requires a "No Evaluation" data channel for triggering the measurement (must be the first data channel in each case).
   b) Add data channels with data evaluation by script `PEAXACTComponentAnalysis.obs`
3) Modify the scenario script according to instructions in next Section
4) Run the process script in OPUS.
5) In case of errors: PEAXACT runtime errors are logged to the `peaxactAppServer.log` file which is located in directory `APPDATADIR` (See note in Section 2).

4.1.3 Modifying OPUS scenario file

Important Notes

- Set-up the whole OPUS PROCESS scenario first using the OPUS scenario browser.
- Run and test the scenario before making manual modifications to the scenario file.
- Once the scenario script is modified manually, the scenario should not be changed with the OPUS scenario browser anymore because this would overwrite all manual modifications. Again, make sure to finish all steps in the scenario browser first.
- Use the OPUS script editor (`Menu File > Open > *.obs`) to modify the scenario script as follows below. If you copy and paste text from a PDF version of this document, copy each page separately because this will preserve line breaks and also prevents from copying headers and footers.

At the beginning of the script, after `Option Explicit` add:

```
' Added by S-PACT %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Dim pxToolbox
Dim outputStruct
Dim isInitializedPEAXACT
isInitializedPEAXACT = False
' %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

At the beginning of sub-procedure `Form_OnLoad()` add:

```
' Added by S-PACT %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
MsgBox "Starting PEAXACT..."
Dim isOK, logLevel, logFilename
Set pxToolbox = CreateObject("PEAXACT.Toolbox")
Set outputStruct = CreateObject("MWComUtil.MWStruct7.14")
pxToolbox.setLogger 1, isOK, "exception", Null
pxToolbox.initialize 1, isOK
If NOT isOK Then
    MsgBox "Failed to initialize PEAXACT."
Else
    pxToolbox.getLogger 2, logLevel, logFilename
    If logLevel = "OFF" Then
        MsgBox "PEAXACT initialized." & Chr(13) & "Logging is OFF"
    Else
        MsgBox "PEAXACT initialized." & Chr(13) & "Logging to " & logFilename
    End If
    pxToolbox.addModel 1, isOK, "<modelFilename>"
    If NOT isOK Then MsgBox "Failed to add model." Else isInitializedPEAXACT = True
End If
' %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Customize <modelFilename> to load your models

- Substitute <modelFilename> with the full path and filename of a PEAXACT model! For instance, the line would then read:

```
pxToolbox.addModel 1, isOK, "C:\models\cyclohexaneModel.pxm"
```

- If you want to add more models, duplicate the `pxToolbox.addModel` line.

At the very end of the script, add:

```
' Added by S-PACT %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Function PEAXACTAnalysis(ByVal typeOfAnalysis, ByVal block, ByVal Id)
    Dim vntResult, iPoint, nPoints, firstX, lastX, isOK, path, file, URI
    Dim xData(), yData(), yDataRow(0), varargin(2), fieldName, result
    result = -1 ' initialize result
    If isInitializedPEAXACT Then
        ' set some options and read data
        vntResult = Form.OpusRequest("BINARY")
        vntResult = Form.OpusRequest("FLOAT_MODE")
        vntResult = Form.OpusRequest("FLOATCONV_MODE ON")
        vntResult = Form.OpusRequest("DATA_POINTS")
        vntResult = Form.OpusRequest("READ_FROM_BLOCK " & block)
        vntResult = Form.OpusRequest("READ_PARAMETER PAT")
        path = split(vntResult, chr(10))(1)
        vntResult = Form.OpusRequest("READ_PARAMETER NAM")
        file = split(vntResult, chr(10))(1)
        vntResult = Form.OpusRequest("READ_PARAMETER NPT")
        nPoints = split(vntResult, chr(10))(1)
        vntResult = Form.OpusRequest("READ_PARAMETER FXV")
        firstX = split(vntResult, chr(10))(1)
        vntResult = Form.OpusRequest("READ_PARAMETER LXV")
        lastX = split(vntResult, chr(10))(1)
        ' create xData
        ReDim xData(nPoints-1, 0) 'column vector
        For iPoint = 0 To nPoints-1
            xData(iPoint, 0) = CDbl(firstX + iPoint * (lastX-firstX)/(nPoints-1))
        Next
        ' read yData and convert to 2D array
        vntResult = Form.OpusRequestData("READ_DATA", yDataRow)
        ReDim Preserve yData(nPoints-1, 0)
        For iPoint = 0 To nPoints-1
            yData(iPoint, 0) = CDbl(yDataRow(iPoint+1)) 'yDataRow starts at index 1
        Next
        ' add data set
        URI = path & chr(92) & file & "#" & block & "-1"
        varargin(0) = xData : varargin(1) = yData : varargin(2) = "clear"
        pxToolbox.addDataSet 1, isOK, URI, varargin
        ' analysis
        If isOK Then
            Select Case UCase(typeOfAnalysis)
                Case "INTEGRATION"
                    pxToolbox.analysisIntegration 2, isOK, outputStruct, Id
                    fieldName = "A"
                Case "COMPONENTFITTING"
                    pxToolbox.analysisComponentFitting 2, isOK, outputStruct, Id
                    fieldName = "w"
                Case "PREDICTION"
                    pxToolbox.analysisPrediction 2, isOK, outputStruct, Id
                    fieldName = "x"
                Case "PREDICTIONOUTLIERPLS"
                    pxToolbox.analysisPrediction 2, isOK, outputStruct, Id
                    fieldName = "mahalanobisDistanceOutlierPValue"
                Case "PREDICTIONOUTLIERIHM"
                    pxToolbox.analysisPrediction 2, isOK, outputStruct, Id
                    fieldName = "RMSResidualsOutlierPValue"
                Case Else : MsgBox "Invalid typeOfAnalysis: " & typeOfAnalysis
            End Select
            If isOK Then result = outputStruct.Item(1, fieldName)
        End If
    End If
    PEAXACTAnalysis = vbLf & vbLf & CStr(result) ' set output
End Function
' %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Search and replace placeholder script

- Press CTRL+F3 to open the text search dialog
- Search for `PEAXACTComponentAnalysis.obs` (ignore any matches found in the first line)
- A matching line should start with `vntResult = Form.OpusRequest("VBScript`
- Replace the whole line by

```
' Modified by S-PACT %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
vntResult = PEAXACTAnalysis("<typeOfAnalysis>", "<block>", "<componentName>")
' %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

- Substitute `<typeOfAnalysis>` with one of the following types:
  - `integration` – calculation of peak area; requires an integration model
  - `componentFitting` – calculation of component weight; requires a hard model
  - `prediction` – prediction of feature value; requires a calibration model
  - `predictionOutlierPLS` – calculates the probability (p-value) for a spectral outlier towards a PLS model; requires a PLS calibration
  - `predictionOutlierIHM` – calculates the probability (p-value) for a spectral outlier towards an IHM model; requires an IHM calibration
- Substitute `<block>` with the desired file block, e.g. `AB`
- Substitute `<componentName>` depending on your choice of `<typeOfAnalysis>`:
  - `integration`: substitute with the name of an integration range
  - `componentFitting`: substitute with the name of a component model
  - `prediction`, `predictionOutlierPLS`, `predictionOutlierIHM`: substitute with the name of a calibrated feature. Be careful not to accidentally use names of integration ranges or component models. Calibrated feature names can be found in the model summary report:

CALIBRATION SUMMARY	
Calibration method	IHM ratiometric
Features	3
Feature names	Cyclohexane, Dioxane, Toluene
Linked component models	Cyclohexane, Dioxane, Toluene
Training samples	8

- For instance, the line would now read:

```
vntResult = PEAXACTAnalysis("prediction", "AB", "Cyclohexane")
```

> Note: `<componentName>` can also be the component's index. The index is consecutively numbered across all added models. For instance, the line would read:
>
> `vntResult = PEAXACTAnalysis("prediction", "AB", 1)`
>
> Do not enclose the index in double quotes! Use the index instead of the name when multiple components have identical names.

- Repeat this step until all occurrences of `PEAXACTComponentAnalysis.obs` are replaced.

4.2 HoloPro

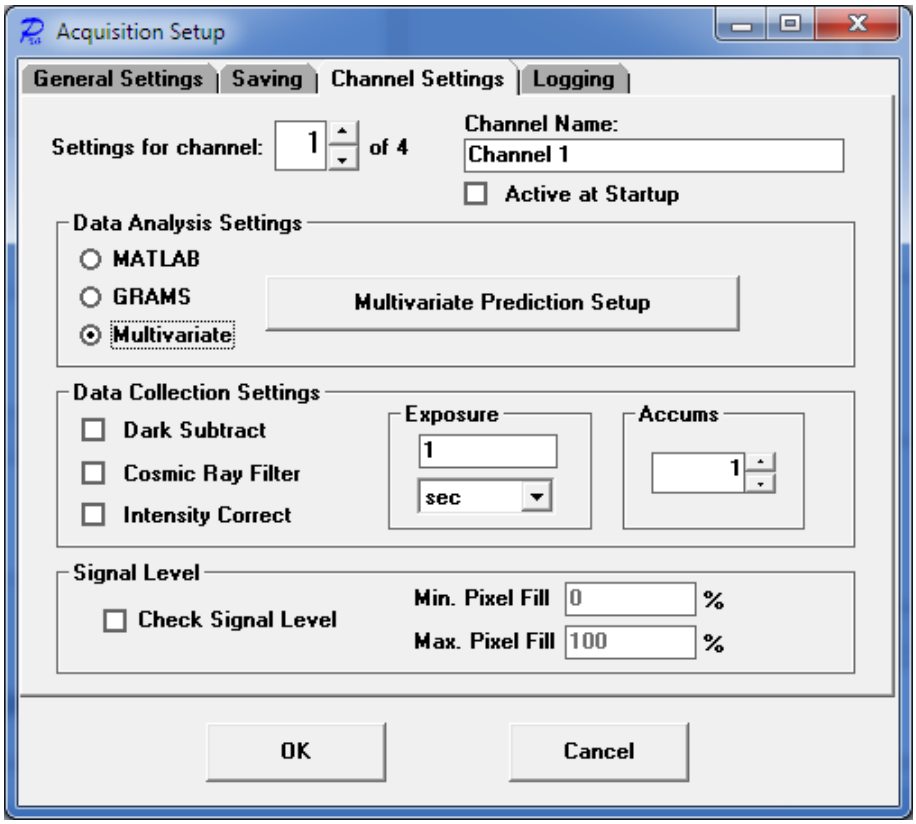
4.2.1 Prerequisites

Software Requirements

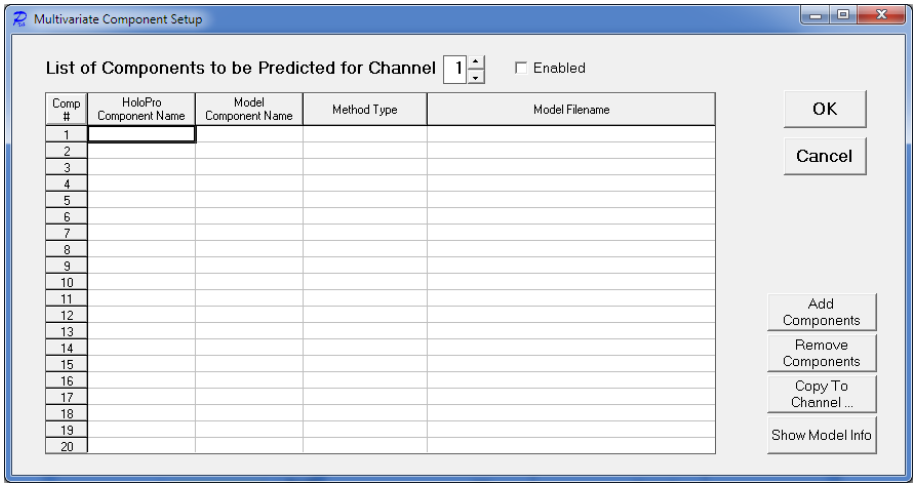
- HoloPro 3.2.0.6 or higher (expected to be installed in directory `C:\Holopro`)
- PEAXACT 3.5 or higher. A 32-bit installation is required.

4.2.2 Configuration

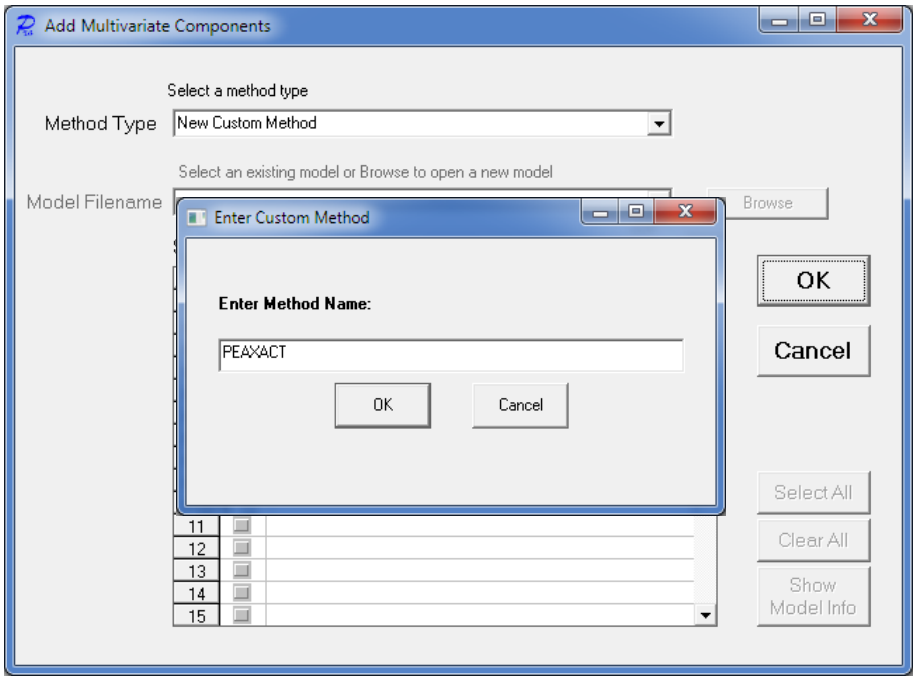
1) Run the diagnosis program first to test whether the PEAXACT Application Server (COM API) is installed and registered correctly.
2) Start HoloPro and open the **Channel Settings** (menu **Settings > Acquisition Setup**)



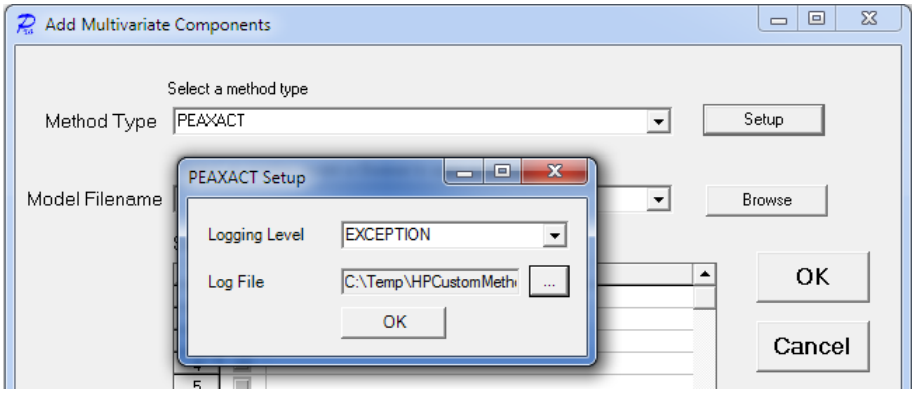
3) Tick **Multivariate** in the Data Analysis Settings Panel, then click the **Multivariate Prediction Setup** button
4) At the top of the In the next window, select a channel, then click the **Add Components** button



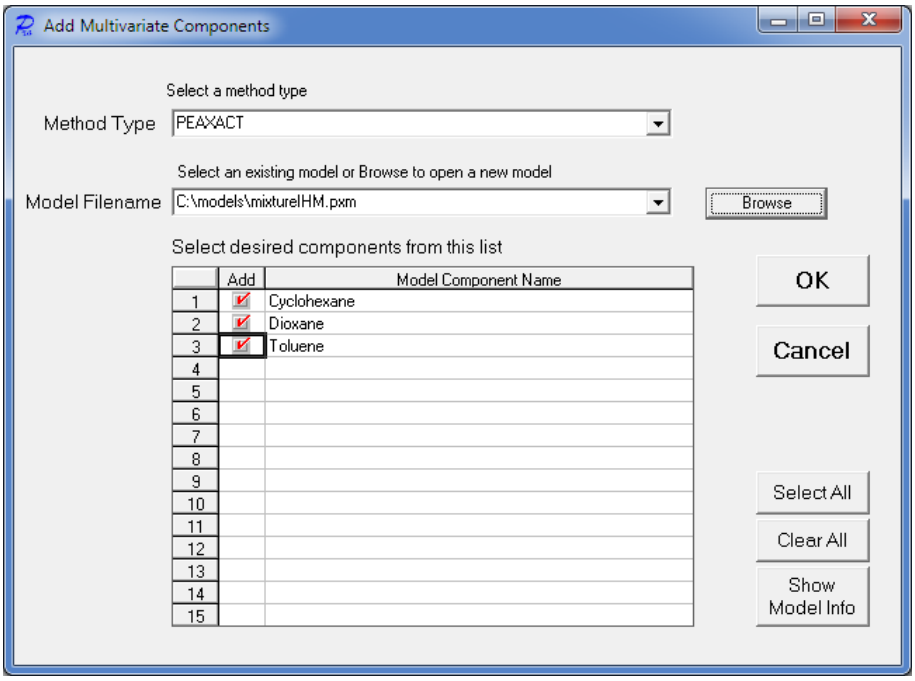
5) From the **Method Type** list select **New Custom Method** and enter `PEAXACT` (or select `PEAXACT` if it has already been added before). It may take a while until the PEAXACT is added to the list, but from then on it will be available permanently.



Note: Optionally, after step 5, you may click the **Setup** button in order to change PEAXACT logging options. In case of unexpected errors you should change the logging level to DEBUG and read the log file.



6) **Browse** for a calibrated model file and select components. Close with **OK**.



7) You can add more components from other models to the same channel, or you can add components to other channels by repeating steps 4 to 6.

> Note: In step 6, you can also browse for a PEAXACT session file in order to load multiple models from the session.

8) After closing all setup windows with **OK** you may start measuring. The prediction of feature values takes place after each measurement. Results will be displayed in the main window of HoloPro.

5 TROUBLE SHOOTING

Problems with the COM API

Symptoms
You cannot access the PEAXACT COM API from your third-party application.

Resolution
In case of any problems with the PEAXACT COM API you should try the following

1) Run the diagnosis program

   After starting the program it performs several tests. In case of errors a possible solution is suggested. You have to fix all problems before you can use the interface correctly.

2) Under some circumstances the diagnosis program crashes (throwing a Windows error) when the COM DLL is registered incorrectly. If this happens, you have to register the DLL manually by executing the file

   ```
   INSTALLDIR\DLL\COM\register.bat
   ```

   Note that administrator privileges are required to execute the file. Afterwards, run the diagnosis again.

Problems with the HoloPro Custom Interface

Symptoms
You receive an error when trying to add PEAXACT as new Custom Method in HoloPro.

Resolution
In case of problems with the HoloPro Custom Interface you should try the following

1) Run the diagnosis program

   After starting the program it performs several tests. In case of errors a possible solution is suggested. You have to fix all problems before you can use the interface correctly.

2) Under some circumstances the diagnosis program crashes (throwing a Windows error) when the custom interface DLL is registered incorrectly. If this happens, you have to register the DLL manually by executing the file

   ```
   INSTALLDIR\DLL\HoloPro\register.bat
   ```

   Note that administrator privileges are required to execute the file. Afterwards, run the diagnosis again.