PEAXACT AppServer
User Manual

Version 5.10
2024-10-30

S-PACT GmbH
Burtscheider Str. 1
52064 Aachen
Germany

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

CONTENTS

1 QUICK START

1.1 What is PEAXACT AppServer?

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

- .NET Framework assembly
- COM component (Component Object Model)

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

In addition, the AppServer provides ready-to-use custom interfaces for third-party software.

1.2 Getting Help

User Manual

This user manual documents a certain version of the PEAXACT AppServer. You can find the version number and release 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 in subdirectory `Help` of the PEAXACT installation directory.

Technical Support

Technical Support can be contacted by:

- E-mail to support@s-pact.com
- Web form at www.s-pact.com/support

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

1.3 Installation & License Activation

1.3.1 System Requirements

- 64-bit version of Microsoft Windows 7 SP1 or Windows 10
- Any Intel or AMD x64 processor
- 5 GB of disk space
- 4 GB RAM
- Microsoft .NET Framework 4.5 or newer

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 AppServer can be installed and operated on a given number of designated computers, provided it is only operated locally (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 www.peaxact.com/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 number. If you do not have a license yet you can get a free trial license or purchase a license after installation.

Step 2: Install PEAXACT

- Download the PEAXACT Installer from www.peaxact.com/download

> Note: The installer's file name is `PeaxactInstaller_<major>.<minor>_win64.exe`. Different major versions can be installed side-by-side, e.g., versions 5 and 4. The installer upgrades earlier installations of the same major version.

Online Installation

- 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 must download additional runtime packages in advance from www.peaxact.com/runtime
- Save all installer files to a folder on a portable drive. Do not rename files.
- At the offline computer, run the PEAXACT Installer 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 www.peaxact.com/whatsnew for further upgrade steps.
- Consider configuring the AppServer service.

1.3.4 License Activation

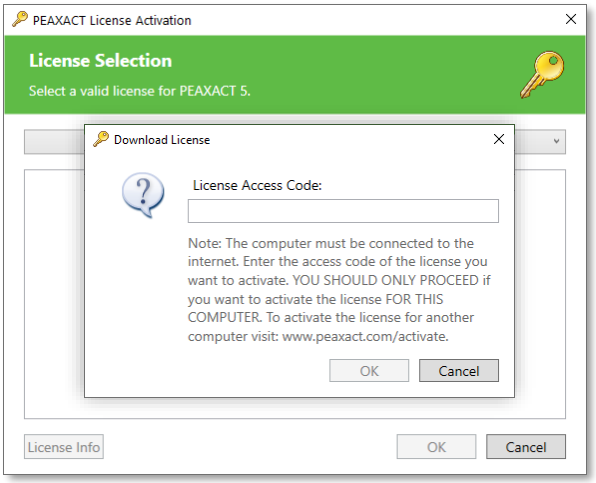
Note: A **license access code** may be required for activation. Codes are provided to end-users or designated license administrators after a license purchase or trial request.
Note: If you perform the activation with administrator privileges, licenses will be activated **per-machine**, i.e., for all Windows users. Otherwise, licenses will be activated **per-user**, i.e., for the logged-on user. Per-machine takes precedence over per-user.

Select **PEAXACT 5 > Activate PEAXACT** from the Windows start menu to open the License Activation Dialog. Then select the **PEAXACT AppServer** product.

Online Activation

To activate PEAXACT over the internet:

- Select **Download License...** from the drop-down list.

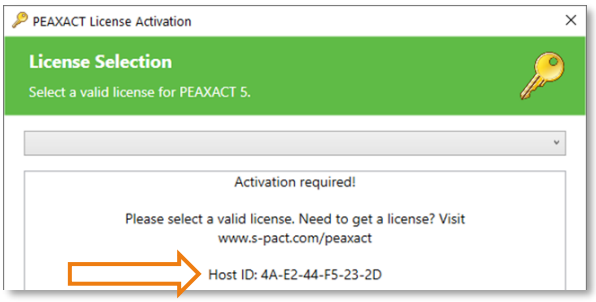


- Enter your license access code and click **OK**. Then close the dialog.

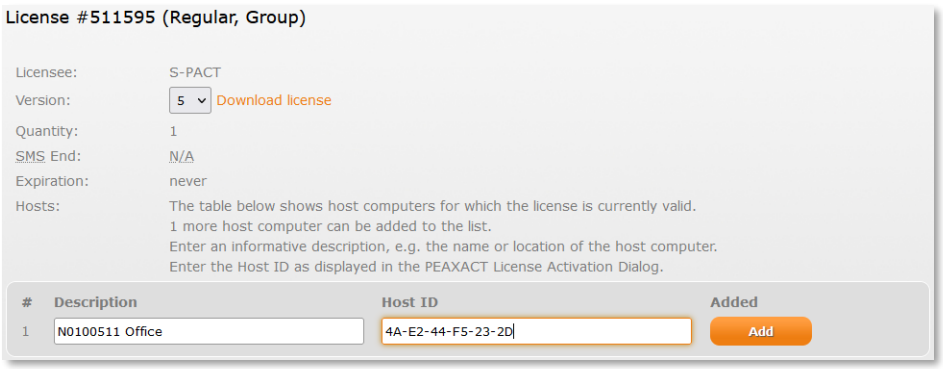
Offline Activation

To activate PEAXACT on a computer without internet you must download the license using another device.

- Make a note of the **Host ID** displayed in the License Activation Dialog.



- On a device with internet, visit www.peaxact.com/activate.
- Sign in to the License Center with your license access code.



- Enter a description (e.g., the computer name) and the **Host ID**, then click **Add**.
- Click **Download license** and save the license file to a portable device.
- In the License Activation Dialog, select **Import License…** and load the license file. Then close the dialog.

Activation per API

The PEAXACT AppServer can also be activated programmatically using the application programming interface.

1.3.5 Configuration of the AppServer service

The AppServer is hosted by a Windows service named `PeaxactAppServerService5 (PEAXACT AppServer 5)` which gets installed by the PEAXACT Installer and automatically starts with Windows. The service can be started with optional parameters.

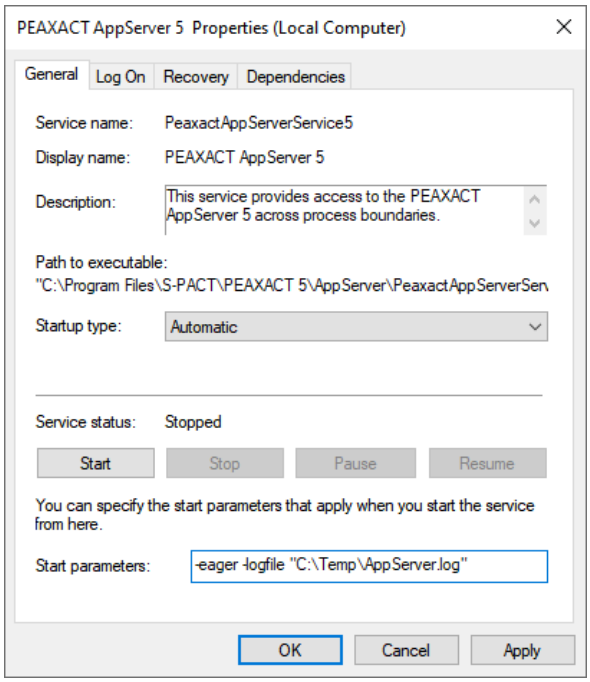
Start Parameters

`-eager`	Fully initializes the AppServer when the service gets started. Without the parameter, the AppServer gets initialized when it is used for the first time, then causing a latency of 30 to 60 seconds. Consider using this start parameter if you are using the AppServer regularly and want to avoid the latency.

`-logfile "<filePath>"`	Enables file logging. `<filePath>` must be the path to a log file. Use this start parameter for debugging purposes only!

One-time Start with Parameters

From the Windows start menu run `services.msc` to open the Windows Services Console, then double-click on **PEAXACT AppServer 5**.



First click **Stop**, then edit **Start parameters**, then click **Start** followed by **OK**. The parameters entered here are not saved; they are passed to the service on a one-time basis.

Always Start with Parameters

From the Windows start menu run `regedit` to open the Windows Registry Editor. Browse to `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\PeaxactAppServerService5`, then double click the `ImagePath` subkey and edit the string value to include start parameters, e.g.,
`"C:\Program Files\S-PACT\PEAXACT 5\AppServer\Service\PeaxactAppServerService.exe" -eager`
Restart the service for the changes to take effect. The service will now always be started with these parameters, e.g., when the service is started automatically on Windows startup.

1.4 Before You Start

Before you access the AppServer 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 5 > Diagnosis of PEAXACT AppServer**.

The diagnosis program performs some tests and suggests possible solutions in case of problems. You should fix all problems before you proceed. Typical problems include:

- MATLAB Runtime is not installed correctly.
- Required DLL files are not registered correctly.
- The AppServer service is not running.

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

2 APPLICATION PROGRAMMING INTERFACE (API)

2.1 .NET API

The .NET API is a set of classes contained in a design-time assembly you would compile and link against when building your own managed assemblies. The assembly file is located at

```
<INSTALLPATH>\AppServer\NET4.5\PeaxactAppServer.dll
```

Before you can use it, you need to reference the assembly in your Visual Studio project. Add the following `<ItemGroup>` block to your project file and replace `<INSTALLPATH>` with the path of the PEAXACT installation directory. This adds a reference to the assembly and will copy dependent native libraries to the output directory when the project is compiled.

```
<ItemGroup>
  <Reference Include="PeaxactAppServer">
    <HintPath><INSTALLPATH>\AppServer\NET4.5\PeaxactAppServer.dll</HintPath>
  </Reference>
  <None Include="<INSTALLPATH>\AppServer\NET4.5\grpc_csharp_ext.*.dll">
    <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
  </None>
</ItemGroup>
```

Target Framework

The assembly targets .NET Framework 4.5 but is compatible with .NET Core and .NET 5+.

Dependencies

The .NET API requires MATLAB Runtime 9.6 to be installed. Also, the Windows service that hosts the AppServer must be installed and running. The service is installed and started automatically by the PEAXACT Installer, or can be installed and started manually by executing the files:

```
<INSTALLPATH>\AppServer\Service\InstallService.vbs
<INSTALLPATH>\AppServer\Service\StartService.vbs
```

Deployment

When packaging your application, include all files from the directories:

`<INSTALLPATH>\AppServer\NET4.5`

`<INSTALLPATH>\AppServer\Service`

When deploying you application, make sure to install MATLAB Runtime 9.6 and install `<INSTALLPATH>\AppServer\Service\PeaxactAppServerService.exe` as a Windows service.

As an alternative to packaging and deploying the AppServer with your own application, simply use the PEAXACT Installer to set up the AppServer. In this case, consider binding your application

to the newest installed version of the AppServer, allowing you and users of your application to update the AppServer independently of your application (see below).

Backward Compatibility

New versions of the .NET API will be backward compatible. Therefore, it is recommended that you do not bind your application to a specific version of the AppServer assembly but instead bind dynamically to the newest version installed on the target computer. See Section 2.3.5 for a programming example.

Support for Asynchronous Analyses

The .NET API provides methods to perform analyses asynchronously. However, note that analyses are executed internally by the MATLAB Runtime which is single-threaded, i.e., if you run multiple asynchronous operations in parallel, they still get executed one after another.

Exception Handling

The .NET API throws exceptions of predefined .NET exception types with specific error messages. Use exception handling code (try/catch blocks) appropriately for all method calls.

2.1.2 Analyzer Class

The `Analyzer` is the main class of the API, representing an isolated environment where analyses execute. The `Analyzer` provides methods to add models and use them to perform analyses of samples.

Notes

When creating the first `Analyzer` instance, expect a latency of 30 to 60 seconds for the initialization of the AppServer. Once initialized, no further latency is to be expected. Consider configuring the AppServer service to initialize the AppServer when the service starts during Windows startup.

Consider creating `Analyzer` objects once at the beginning of your application for each model (or set of models) and keep the objects alive for as long as you want to perform analyses with them. This is because adding models involves loading potentially large files and might be slow.

Constructors

`Analyzer()`	Initializes a new instance of the `Analyzer` class with `DefaultOptions`.
`Analyzer(AnalyzerOptions options)`	Initializes a new instance of the `Analyzer` class with custom options. `options`: The options.

Properties

`DefaultOptions : AnalyzerOptions`

Gets or sets default options to be used when calling the `Analyzer()` constructor without custom options.

`Models : IEnumerable<Model>`

Gets the models added by `AddModel()`.

Methods

`AddModel(string filePath) : void`

Adds a model.
`filePath`: The path to a PEAXACT model file (extension PXM) to be added. It can also be the path to a PEAXACT session file (extension PXS) containing any number of models to be added.

`PerformAnalysisPreview(AnalysisType type) : IEnumerable<AnalysisResult>`

Performs a preview of an analysis, returning the same results as `PerformAnalysis()`, but without values. This is helpful to find out in advance the number and names of results, as well as the hierarchical structure of sub-results. If needed, use the `Traverse()` extension method for `IEnumerable<AnalysisResult>` to convert the hierarchical result sequence into a flat sequence.
`type`: The type of the analysis. Consider using type `Auto` so that it is determined by the added models.

`PerformAnalysis(AnalysisType type, Sample sample) : IEnumerable<AnalysisResult>`

Performs an analysis of a sample and returns `type`-dependent results. The sample gets processed by all added models that match the specified analysis `type`. If needed, use the `Traverse()` extension method for `IEnumerable<AnalysisResult>` to convert the hierarchical result sequence into a flat sequence.
`type`: The type of the analysis. Consider using type `Auto` so that it is determined by the added models.
`sample`: The sample to analyze.

`PerformPeakPicking(Sample sample, PeakPickingOptions options) : PeakPickingResults`

Performs peak picking of a sample, applying pretreatments of the first added model (if any).
`sample`: The sample to analyze.
`options`: The options for peak detection.

`PerformMcr(IEnumerable<Sample> samples, McrOptions options) : McrResults`

Performs Multivariate Curve Resolution of a set of samples, applying pretreatments of the first added model (if any).
`samples`: The collection of samples to analyze.
`options`: Options for the MCR algorithm.

`PerformMcrAsync(IEnumerable<Sample> samples, McrOptions options, CancellationToken cancellationToken, IProgress<int> progress) : Task<McrResults>`	Asynchronously performs Multivariate Curve Resolution of a set of samples, applying pretreatments of the first added model (if any). Returns a task object representing the asynchronous operation. `samples`: The collection of samples to analyze. `options`: Options for the MCR algorithm. `cancellationToken`: The token to monitor for cancellation requests. `progress`: The provider for progress updates.
`PerformHmfa(IEnumerable<Sample> samples, HmfaOptions options) : HmfaResults`	Performs Hard Modeling Factor Analysis of a set of samples, using the first added model. `samples`: The collection of samples to analyze. `options`: Options for the HMFA algorithm.
`PerformHmfaAsync(IEnumerable<Sample> samples, HmfaOptions options, CancellationToken cancellationToken, IProgress<int> progress) : Task<HmfaResults>`	Asynchronously performs Hard Modeling Factor Analysis of a set of samples, using the first added model. Returns a task object representing the asynchronous operation. `samples`: The collection of samples to analyze. `options`: Options for the HMFA algorithm. `cancellationToken`: The token to monitor for cancellation requests. `progress`: The provider for progress updates.

2.1.3 AnalyzerOptions Class

The `AnalyzerOptions` class provides license information to the `Analyzer` class. You could either pass the options to the `Analyzer(AnalyzerOptions)` constructor or assign the options to the static property `Analyzer.DefaultOptions` and use the default constructor (recommended).

Constructors

`AnalyzerOptions()`	Initializes a new instance of the `AnalyzerOptions` class.

Properties

`LicenseFilePath : string`	Gets or sets the path to a PEAXACT license file (extension LIC). Use `null` or an empty string for auto-detection, in which case the Windows Registry is searched for a license file path registered by the PEAXACT License Activation Dialog.
`LicensePassword : string`	Gets or sets the license password required for protected licenses.

2.1.4 Model Class

The `Model` is a read-only object providing model specifications. It is useful to determine the kinds of analyses a model can be used for:

- All models can be used for `Analyzer.PerformPeakPicking()`, `Analyzer.PerformMcr()`, and `Analyzer.PerformAnalyis` with `AnalysisType.Auto`.
- Models with `IntegrationModelComponentNames` are suitable for `Analyzer.PerformAnalysis()` with `AnalysisType.Integration`.
- Models with `HardModelComponentNames` are suitable for `Analyzer.PerformAnalysis()` with `AnalysisType.ComponentFitting` or `Analyzer.PerformHmfa`.
- Models with `CalibrationModelComponentNames` are suitable for `Analyzer.PerformAnalysis()` with `AnalysisType.Prediction`.
- Models with `ClassificationModelComponentNames` are suitable for `Analyzer.PerformAnalysis()` with `AnalysisType.Identification`.
- Models with `CustomModelComponentNames` are suitable for `Analyzer.PerformAnalysis()` with `AnalysisType.Custom`.

Properties

`Path : string`	Gets the model file path.
`Checksum : string`	Gets the SHA-1 checksum of the model file. *Note: This member was added in v5.7.*
`Signature : string`	Gets information about the model's digital signature.
`Description : string`	Gets the description of the model as provided by the creator of the model.
`IntegrationModelComponentNames : string[]`	Gets the names of Integration Model Components.
`HardModelComponentNames : string[]`	Gets the names of Hard Model Components.
`CalibrationModelComponentNames : string[]`	Gets the names of Calibration Model Components.
`ClassificationModelComponentNames : string[]`	Gets the names of Classification Model Components.
`ClassificationModelClassNames : string[]`	Gets the names of the Classification Model classes.
`CustomModelComponentNames : string[]`	Gets the names of Custom Model Components.

2.1.5 Sample Class

The `Sample` represents measured data of a physical sample. The `Sample` object at least contains the file path to a measured spectrum or chromatogram. It may also contain the x-values and y-values of the measured signal itself as well as names and values of additional features associated with the sample, e.g., timestamp, temperature, or meta information.

The `Sample` class uses a URI (uniform resource identifier) to identify a resource in a file. The URI is a string that consists of an absolute or relative file path plus #ID, e.g.: `C:\Data\SpectrumFile.spc#1`. The #ID-part of the URI is optional. It can be used to refer to a certain resource in a file that contains multiple resources. The format of the ID depends on the file type, but typically is just a number. The default is 1.

Constructors

`Sample(string uri)`	Initializes a new instance of the `Sample` class. `uri`: The URI of the sample. It must refer to an existing resource file from which to load x-values and y-values.
`Sample(string uri, double[] x, double[] y)`	Initializes a new instance of the `Sample` class with x-values and y-values. `uri`: The URI of the sample. It could refer to an existing resource file, but when x- and y-values are provided it could also be an arbitrary file path because the file will never be accessed. `x`: The x-values of the sample. `y`: The y-values of the sample. *Note: When* `x` *and* `y` *are* `null`*, values will be read from* `uri`*.*
`Sample(string uri, double[] x, double[] y, Dictionary<string, object> features)`	Initializes a new instance of the `Sample` class with x-values, y-values, and additional features. `uri`: The URI of the sample. `x`: The x-values of the sample. `y`: The y-values of the sample. `features`: The names and values of additional features associated with the sample. Values can be of type `double` or `string`.

`Sample(string uri, double[] x, double[] y, double[] i)`	Initializes a new instance of the `Sample` class with x-values and complex y-values. `uri`: The URI of the sample. It could refer to an existing resource file, but when x- and y-values are provided it could also be an arbitrary file path because the file will never be accessed. `x`: The x-values of the sample. `y`: The real part of complex y-values of the sample. `i`: The imaginary part of complex y-values (`null` = missing). *Note: This member was added in v5.5.*
`Sample(string uri, double[] x, double[] y, double[] i, Dictionary<string, object> features)`	Initializes a new instance of the `Sample` class with x-values, complex y-values, and additional features. `uri`: The URI of the sample. `x`: The x-values of the sample. `y`: The real part of complex y-values of the sample. `i`: The imaginary part of complex y-values (`null` = missing). `features`: The names and values of additional features associated with the sample. Values can be of type `double` or `string`. *Note: This member was added in v5.5.*

Properties

`Uri : string`	Gets the URI of the sample.
`X : double[]`	Gets the x-values of the sample.
`Y : double[]`	Gets the y-values of the sample.
`I : double[]`	Gets the optional imaginary part of complex y-values. *Note: This member was added in v5.5.*
`Features : Dictionary<string, object>`	Gets the names and values of features associated with the sample. Values can be of type `double` or `string`.

2.1.6 AnalysisResult Class

An `AnalysisResult` object represents a result of an analysis. It may contain other results (sub-results) depending on the type of analysis. A sequence of such hierarchical results is returned by `Analyzer.PerformAnalysisPreview()` and `Analyzer.PerformAnalysis()`. The `Traverse()` extension method for `IEnumerable<AnalysisResult>` provides a convenient way to convert the sequence of hierarchical results into a flat sequence, and the `FullName` property represents a convenient alternative over the `Name` property when working with flat sequences.

Properties

`Type : AnalysisResultType`	Gets the result type.
`Name : string`	Gets the result name.
`Value : object`	Gets the numeric or categorical result value. If `IsNumeric` is `true`, the value is of type `double`, otherwise of type `string`.
`IsNumeric : bool`	Gets whether `Value` is numeric (`true`) or categorical (`false`).
`Children : IEnumerable<AnalysisResult>`	Gets the sequence of sub-results.
`FullName : string`	Gets the full name, which consists of the joined names from the top-level result to the result you want, separated by `" - "`. *Note: This member was added in v5.4.*

2.1.7 AnalysisType Enumeration

Specifies the type of analysis performed by `Analyzer.PerformAnalysisPreview()` and `Analyzer.PerformAnalysis()`. Consider using type `Auto` so that it is determined by the added models.

Fields

`Auto = -1`	Analysis as specified by added models, or, if not specified, the first analysis for which a model returns results, in the order: `Custom`, `Identification`, `Prediction`, `ComponentFitting`, `Integration` *Note: This member was added in v5.4.*
`Custom = 0`	Analysis of custom results as configured in a Custom Model.
`Integration = 1`	Integration of the measured signal as configured in an Integration Model.
`ComponentFitting = 2`	Fitting of the measured signal as configured in a Hard Model.
`Prediction = 3`	Prediction of feature values as configured in a Calibration Model.
`Identification = 4`	Identification of class values as configured in a Classification Model.

2.1.8 AnalysisResultType Enumeration

Specifies the type of a result returned by `Analyzer.PerformAnalysisPreview()` and `Analyzer.PerformAnalysis()`.

Note: More result types may be added in the future. Make sure your application can deal with unknown types so that your code will continue to work when new types get added.

Fields

`CustomNumeric = 0`	Numeric value calculated by a Custom Model.
`CustomCategorical = 1`	Categorical value calculated by a Custom Model.
`IntegrationComponentArea = 10`	Component area calculated by an Integration Model.
`ComponentFittingWeight = 20`	Component weight of a fitted Hard Model.
`ComponentFittingArea = 21`	Component area of a fitted Hard Model.
`ComponentFittingRmsResiduals = 22`	RMS residuals of a fitted Hard Model.
`PredictionValue = 30`	Predicted value calculated by a Calibration Model.
`PredictionStandardUncertainty = 31`	Standard uncertainty of the predicted value.
`PredictionExpandedUncertainty = 32`	Expanded uncertainty of the predicted value for a 95% level of confidence.
`PredictionRmsResiduals = 33`	RMS residuals, either of a fitted Hard Model (HM regression) or of a factor model (PLS regression).
`PredictionRmsResidualsOutlierProbability = 34`	Probability for a value of type `PredictionRmsResiduals` being an outlier.
`PredictionMahalanobisDistance = 35`	Mahalanobis distance of a sample to training samples, calculated by a factor model (PLS regression).
`PredictionMahalanobisDistanceOutlierProbability = 36`	Probability for a value of type `PredictionMahalanobisDistance` being an outlier.
`IdentificationClassName = 40`	Identified class name calculated by a Classification Model.

`IdentificationClassNumber = 41`	Identified class number calculated by a Classification Model. The number can be used as a (one-based) index into the combined array of all possible class names from `Analyzer.Models[…].ClassificationModelClassNames`.
`IdentificationClassProbability = 42`	Probability for a value of type `IdentificationClassName` being the actual class.
`IdentificationMahalanobisDistance = 43`	Mahalanobis distance of a sample to training samples, calculated by a Classification Model.
`IdentificationMahalanobisDistanceOutlierProbability = 44`	Probability for a value of type `IdentificationMahalanobisDistance` being an outlier.

2.1.9 PeakPickingOptions Class

Options used by `Analyzer.PerformPeakPicking()`.

Constructor

`PeakPickingOptions()`	Initializes a new instance of the `PeakPickingOptions` class.

Properties

`MinimumPeakHeight : double?`	Gets or sets the minimum height for peak detection. `null` = auto-detect (default).

2.1.10 PeakPickingResults Class

Results returned by `Analyzer.PerformPeakPicking()`.

Properties

`SampleUri : string`	Gets the URI of the analyzed sample.
`MinimumPeakHeight : double`	Gets the minimum height of peaks used for the analysis. This is either the user-specified input value or an auto-detected value.
`PeakIndices : double[]`	Gets one-based indices into `XData` or `YData` of found peaks.

`PeakPositions : double[]`	Gets x-values of found peaks.
`PeakIntensities : double[]`	Gets y-values of found peaks.
`XData : double[]`	Gets x-values of the sample (after pre-treatments if any) used for the analysis.
`YData : double[]`	Gets y-values of the sample (after pre-treatments if any) used for the analysis.

2.1.11 McrOptions Class

Options used by `Analyzer.PerformMcr()`.

Constructors

`McrOptions(int numComponents)`	Initializes a new instance of the `McrOptions` class. `numComponents`: The number of components to identify.

Properties

`NumComponents : int`	Gets or sets the number of components to identify from the sample set.
`C0 : double[,]`	Gets or sets the optional 2D array of initial concentrations. Rows correspond to samples; columns correspond to components. If `C0` is `null`, concentrations are initialized with previous results (if available). If `C0` is an empty array, concentrations are initialized implicitly (reset).
`ToleranceRmse : double`	Gets or sets the criterion for stopping the MCR-ALS algorithm when progress between iterations drops below the tolerance (default = `1e-5`).
`MaxIterations : int`	Gets or sets the criterion for stopping the MCR-ALS algorithm after a maximum number of iterations (default = `100`).
`MaxUnsuccessfulAttempts : int`	Gets or sets the criterion for stopping the MCR-ALS algorithm after a maximum number of unsuccessful iterations (default = `20`).
`IsNonnegativeC : bool`	Gets or sets whether the non-negativity constraint for component concentrations is active (default = `false`).

`IsNonnegativeS : bool`	Gets or sets whether the non-negativity constraint for component spectra is active (default = `false`).
`IsUnimodalC : bool`	Gets or sets whether the unimodality constraint for component concentrations is active (default = `false`).
`IsClosureC : bool`	Gets or sets whether the closure constraint for component concentrations is active (default = `false`).

2.1.12 McrResults Classs

Results returned by `Analyzer.PerformMcr()`.

Properties

`SampleUris : string[]`	Gets the URIs of the analyzed samples.
`ComponentNames : string[]`	Gets the automatically generated names of identified components.
`S : double[,]`	Gets the spectral intensities (y-values) of identified components. Rows correspond to `XData`; columns correspond to `ComponentNames`.
`C : double[,]`	Gets the concentrations of identified components. Rows correspond to `XData`; columns correspond to `ComponentNames`.
`RmsResiduals : double[]`	Gets the root mean square (RMS) spectral residuals for each sample. Residuals are differences between measured and reconstructed signal.
`R2 : double`	Gets the fraction of the variance of the measured signal which is explained by the reconstructed signal.
`XData : double[]`	Gets x-values of the sample (after pre-treatments if any) used for the analysis.

2.1.13 HmfaOptions Class

Options used by `Analyzer.PerformHmfa()`.

Constructors

`HmfaOptions(int numComponents)`	Initializes a new instance of the `HmfaOptions` class. `numComponents`: The number of components to identify.

Properties

`NumComponents : int`	Gets or sets the number of components to identify from the sample set.
`IsClosureC : bool`	Gets or sets whether the closure constraint for component concentrations is active (default = `false`).

2.1.14 HmfaResults Class

Results returned by `Analyzer.PerformHmfa()`.

Properties

`SampleUris : string[]`	Gets the URIs of the analyzed samples.
`ComponentNames : string[]`	Gets the automatically generated names of identified components.
`S : double[,]`	Gets the spectral intensities (y-values) of identified components. Rows correspond to `XData`; columns correspond to `ComponentNames`.
`C : double[,]`	Gets the concentrations of identified components. Rows correspond to `XData`; columns correspond to `ComponentNames`.
`RmsResiduals : double[]`	Gets the root mean square (RMS) spectral residuals for each sample. Residuals are differences between measured and reconstructed signal.
`R2 : double`	Gets the fraction of the variance of the measured signal which is explained by the reconstructed signal.
`XData : double[]`	Gets x-values of the sample (after pre-treatments if any) used for the analysis.

2.2 COM API

The COM API is a set of classes contained in a COM-visible assembly you would use in scripts and applications that are COM-compliant, e.g., VBScript, Visual Basic, Excel, or LabVIEW. The assembly file is located at

```
<INSTALLPATH>\AppServer\COM\PeaxactAppServerCom.dll
```

Before you can use the COM API, COM classes must be registered in the Windows Registry. The classes are registered automatically by the PEAXACT Installer, or can be registered manually by executing the file:

```
<INSTALLPATH>\AppServer\COM\Register.vbs
```

Dependencies

The .NET API requires MATLAB Runtime 9.6 to be installed. Also, the Windows service that hosts the AppServer must be installed and running. The service is installed and started automatically by the PEAXACT Installer, or can be installed and started manually by executing the files:

```
<INSTALLPATH>\AppServer\Service\InstallService.vbs
<INSTALLPATH>\AppServer\Service\StartService.vbs
```

ProgID

The COM API exposes the `IAnalyzer` class interface and the corresponding `Analyzer` class which is the only class you can create objects from. The `Analyzer` class can be referenced by its ProgId.

Version-specific ProgId: `PEAXACT.Analyzer.5`
Version-independent ProgId: `PEAXACT.Analyzer`

In Visual Basic, e.g., you would create a new instance of the `Analyzer` class as follows:

```
Set analyzer = CreateObject("PEAXACT.Analyzer.5")
```

Backward Compatibility

Within the same major version, new minor versions of the COM API will be backward compatible. New major versions of the COM API are subject to changes and may break your application. Therefore, it is highly recommended that you bind your application to a specific version of the AppServer DLL. You can do this by referencing the version-specific ProgId.

Dynamic Binding vs. Static Binding

The AppServer DLL only allows for dynamic binding (late binding) to assure that your application does not break when new methods or properties are added to the API in the future.

Exception Handling

The COM API throws exceptions with specific error messages. Use exception handling code appropriately for all method calls.

2.2.2 Analyzer Class

The `Analyzer` is the main class of the API, representing an isolated environment where analyses execute. The `Analyzer` provides methods to add models and use them to perform analyses of samples.

Notes

When creating the first `Analyzer` instance, expect a latency of 30 to 60 seconds for the initialization of the AppServer. Once initialized, no further latency is to be expected. Consider configuring the AppServer service to initialize the AppServer when the service starts during Windows startup.

Consider creating `Analyzer` objects once at the beginning of your application for each model (or set of models) and keep the objects alive for as long as you want to perform analyses with them. This is because adding models involves loading potentially large files and might be slow.

Constructors

`Analyzer()`	Initializes a new instance of the `Analyzer` class.

Properties

`Models : Model[]`	Gets the models added by `AddModel()`.

Methods

`AddModel(string filePath) : void`

Adds a model.
`filePath`: The path to a PEAXACT model file (extension PXM) to be added. It can also be the path to a PEAXACT session file (extension PXS) containing any number of models to be added.

`CreateSample(string uri, double[] x, double[] y, params object[] features) : Sample`

Returns a new instance of a `Sample` object.
`uri`: The URI of the sample. If `x` and `y` are `null`, it must refer to an existing resource file from which to load x-values and y-values. Otherwise, it could be an arbitrary file path.
`x`: The numeric array of x-values, or `null` to read from `uri`.
`y`: The numeric array of y-values, or `null` to read from `uri`.
`features`: Optional array of alternating names and values of features associated with the sample. Names must be of type `string`; values can be of type `double` or `string`.

`PerformAnalysisPreview(string analysisType) : AnalysisResult[]`

Performs a preview of an analysis, returning the same results as `PerformAnalysis()`, but without values. This is helpful to find out in advance the number and names of results, as well as the hierarchical structure of sub-results.
`analysisType`: The name of the analysis type. See `AnalysisType` for a list of possible names.

`PerformAnalysis(string analysisType, Sample sample) : AnalysisResult[]`
Performs an analysis of a sample and returns `analysisType`-dependent results. The sample gets processed by all added models that match the specified `analysisType`.
`analysisType`: The name of the analysis type. See `AnalysisType` for a list of possible names. Consider using type `"Auto"` so that it is determined by the added models.
`sample`: The sample to analyze.

`PerformPeakPicking(Sample sample, object minimumPeakHeight) : PeakPickingResults`
Performs peak picking of a sample, applying pretreatments of the first added model (if any).
`sample`: The sample to analyze.
`minimumPeakHeight`: The minimum height for peak detection (`null` = auto-detect = default).

`PerformMcr(Sample[] samples, int numComp, params object[] options) : McrResults`
Performs Multivariate Curve Resolution of a set of samples, applying pretreatments of the first added model (if any).
`samples`: The collection of samples to analyze.
`numComp`: The number of components to identify from the sample set.
`options`: Additional options for the MCR algorithm as name/value-pairs. Names can be: `C0`, `ToleranceRmse`, `MaxIterations`, `MaxUnsuccessfulAttempts`, `IsNonnegativeC`, `IsNonnegativeS`, `IsUnimodalC`, `IsClosureC`. See `McrOptions` for the meaning and appropriate values.

`PerformHmfa(Sample[] samples, int numComp, params object[] options) : HmfaResults`
Performs Hard Modeling Factor Analysis of a set of samples, using the first added model.
`numComp`: The number of components to identify from the sample set.
`options`: Additional options for the HMFA algorithm as name/value-pairs. Names can be: `IsClosureC`. See `HmfaOptions` for the meaning and appropriate values.

2.2.3 Model Class

This class is equivalent to the `Model` class of the .NET API.

2.2.4 Sample Class

The `Sample` represents measured data of a physical sample. The `Sample` object at least contains the file path to a measured spectrum or chromatogram. It may also contain the x-values and y-values of the measured signal itself as well as names and values of additional features associated with the sample, e.g., timestamp, temperature, or meta information.

The `Sample` class uses an URI (uniform resource identifier) to identify a resource in a file. The URI is a string that consists of an absolute or relative file path plus #ID, e.g.: `C:\Data\SpectrumFile.spc#1`. The #ID-part of the URI is optional. It can be used to refer to a certain resource in a file that contains multiple resources. The format of the ID depends on the file type, but typically is just a number. The default is 1.

Constructors

See `Analyzer.CreateSample()`.

Properties

`Uri : string`	Gets the URI of the sample.
`X : double[]`	Gets the x-values of the sample.
`Y : double[]`	Gets the y-values of the sample.
`Features : object[]`	Gets the array of alternating names and values of features associated with the sample.

2.2.5 AnalysisResult Class

An `AnalysisResult` object represents a result of an analysis that may contain other results (sub-results) depending on the type of analysis. A sequence of such hierarchical results is returned by `Analyzer.PerformAnalysisPreview()` and `Analyzer.PerformAnalysis()`.

Properties

`Type : int`	Gets the result type. See `AnalysisResultType` for a list of possible result type numbers.
`Name : string`	Gets the result name.
`Value : object`	Gets the numeric or categorical result value. If `IsNumeric` is `true`, the value is of type `double`, otherwise of type `string`.
`IsNumeric : bool`	Gets whether `Value` is numeric (`true`) or categorical (`false`).
`Children : AnalysisResult[]`	Gets the array of sub-results.
`FullName : string`	Gets the full name, which consists of the joined names from the top-level result to the result you want, separated by `" - "`.

2.2.6 PeakPickingResults Class

This class is equivalent to the `PeakPickingResults` class of the .NET API.

2.2.7 McrResults Class

This class is equivalent to the `McrResults` class of the .NET API.

2.2.8 HmfaResults Class

This class is equivalent to the `HmfaResults` class of the .NET API.

2.3 Programming Examples

2.3.1 Using the .NET API in C#

This example demonstrates how to use the .NET API in C#. The program uses a model, displays a preview of available result names, performs the analysis on a measured sample, and displays result values.

```
using System;
using System.Linq;
using S_PACT.PEAXACT;

namespace Examples
{
    class PerformAnalysisExample
    {
        private Analyzer analyzer;
        private AnalysisType analysisType;

        static void Main(string[] args)
        {
            new PerformAnalysisExample().Run();
            Console.WriteLine("Press any key to continue.");
            Console.ReadKey();
        }

        private void Run()
        {
            // Do things that only need to be done once.
            Initialize();
            // Analyze samples.
            Analyze();
        }

        private void Initialize()
        {
            // Create a new analyzer with options.
            Analyzer.DefaultOptions = new AnalyzerOptions()
            {
                LicenseFilePath = @"C:\license.lic"
            };
            analyzer = new Analyzer();

            // Add one or more models to the analyzer.
            analyzer.AddModel(@"C:\user\model.pxm");
            //analyzer.AddModel(@"C:\user\model2.pxm");

            // Define the analysis type.
            analysisType = AnalysisType.Auto;

            // Optional: Preview names of analysis results.
            var results = analyzer.PerformAnalysisPreview(analysisType);
            // "results" is a sequence of hierarchically structured objects.
            // The first level of results corresponds to components (main results).
            // Each main result may have none to multiple levels of sub-results.
            if (!results.Any()) throw new Exception("Models provide no results.");
            // Loop through main results.
            foreach (var mainResult in results)
            {
                Console.WriteLine(mainResult.Name);
                // Traverse all levels of sub-results (depth-first).
                foreach (var subResult in mainResult.Children.Traverse())
```

```
                {
                    // FullName consists of all names from mainResult to subResult.
                    Console.WriteLine(subResult.FullName);
                }
            }
        }

        private void Analyze()
        {
            // Get the sample to analyze.
            Sample sample = CreateSample();
            // Analyze the sample.
            var results = analyzer.PerformAnalysis(analysisType, sample);
            // Traverse all results, display values.
            foreach (var result in results.Traverse())
            {
                // Numeric or categorical result value?
                if (result.IsNumeric) Console.WriteLine((double)result.Value);
                else Console.WriteLine((string)result.Value);
            }
        }

        private Sample CreateSample()
        {
            // This method creates a Sample object from known x and y values.
            int nx = 1000; // e.g., 1000 data points
            double[] x = new double[nx]; // e.g., wavenumbers
            double[] y = new double[nx]; // intensities

            // Populate x and y.
            // ...

            // Create the sample.
            // The URI can be arbitrary when x and y are provided.
            return new Sample("dummy.xyz", x, y);
        }
    }
}
```

2.3.2 Using the .NET API in Python

This example demonstrates how to use the .NET API in Python. The content of the example is identical to that from the previous section.

> Note: This example requires `pythonnet`. See https://pypi.org/project/pythonnet.

```
import clr
clr.AddReference('C:\Path\To\PeaxactAppServer.dll')
from S_PACT.PEAXACT import Analyzer, AnalyzerOptions, AnalysisType, Sample

class PerformAnalysisExample:
    def run(self):
        # Do things that only need to be done once.
        self.initialize()
        # Analyze samples.
        self.analyze();

    def initialize(self):
        # Create a new analyzer with options.
        options = AnalyzerOptions()
        options.LicenseFilePath = 'C:\license.lic'
        Analyzer.DefaultOptions = options
        self.analyzer = Analyzer()
```

```
        # Add one or more models to the analyzer.
        self.analyzer.AddModel('C:\user\model.pxm');
        #self.analyzer.AddModel('C:\user\model2.pxm');

        # Define the analysis type.
        self.analysis_type = AnalysisType.Auto;

        # Optional: Preview names of analysis results.
        results = self.analyzer.PerformAnalysisPreview(self.analysis_type)
        # "results" is a sequence of hierarchically structured objects.
        # The first level of results corresponds to components (main results).
        # Each main result may have none to multiple levels of sub-results.
        if not results:
            raise Exception('Models provide no results.');
        # Loop through main results.
        for mainResult in results:
            print(mainResult.Name)
            # Traverse all levels of sub-results (depth-first).
            for subResult in traverse(mainResult.Children):
                # FullName consists of all names from mainResult to subResult.
                print(subResult.FullName)

    def analyze(self):
        # Get the sample to analyze.
        sample = self.create_sample()

        # Analyze the sample.
        results = self.analyzer.PerformAnalysis(self.analysis_type, sample)

        # Traverse all results, display values.
        for result in traverse(results):
            # Numeric or categorical result value?
            if result.IsNumeric:
                print(result.Value) # result.Value is a float
            else:
                print(result.Value) # result.Value is a string

    def create_sample(self):
        # This method creates a Sample object from known x and y values.
        x = []
        y = []

        # Populate x and y.
        # ...

        # Create the sample.
        # The URI can be arbitrary when x and y are provided.
        return Sample('dummy.xyz', x, y)

# Converts a hierarchical sequence into a flat sequence (depth-first traversal).
def traverse(results):
    combined_results = []
    for result in results:
        combined_results += [result] + traverse(result.Children)
    return combined_results

if __name__ == '__main__':
    PerformAnalysisExample().run()
```

2.3.3 Using the COM API in VB Script

This example demonstrates how to call the PEAXACT AppServer from Visual Basic Script (VBS). VBS only supports the COM API. The script uses an Integration Model to calculate component areas of a gas spectrum and displays results.

```
' Create a new analyzer.
Set analyzer = CreateObject("PEAXACT.Analyzer.5")

' Add a model to the analyzer.
analyzer.AddModel "c:\model\syngas.pxm"

' Create the sample to analyze.
Set sample = analyzer.CreateSample("C:\data\syngas.csv#1", null, null)

' Perform the integration according to model specifications.
' Note that (sample) is enclosed in parentheses to pass it "byval".
results = analyzer.PerformAnalysis("Integration", (sample))

' Display results.
For Each result In results
    Wscript.Echo result.Name & " = " & result.Value
Next
```

2.3.4 Using the .NET API in MATLAB

This example demonstrates how to call the .NET API from MATLAB (R2009a or newer). The script uses an Integration Model to calculate component areas of a gas spectrum and displays results.

```
% Load the AppServer DLL
appServerPath = 'C:\Program Files\S-PACT\PEAXACT 5\AppServer\NET4.5';
NET.addAssembly(fullfile(appServerPath, 'PeaxactAppServer.dll'));
import S_PACT.PEAXACT.*;

% Create a new analyzer.
analyzer = Analyzer();

% Add a model to the analyzer.
analyzer.AddModel('c:\model\syngas.pxm');

% Create the sample to analyze.
sample = Sample('C:\data\syngas.csv#1');

% Perform the integration according to model specifications.
results = analyzer.PerformAnalysis(AnalysisType.Integration, sample)

% Display results.
for iResult = 0:results.Count - 1
    result = results.Item(iResult);
    fprintf('%s = %f\n', result.Name, result.Value);
end
```

2.3.5 Dynamically bind to future versions of the AppServer assembly (.NET API)

This example demonstrates how you could make your.NET application independent of a specific version of the AppServer and instead bind to the newest AppServer assembly installed on the target computer. This is possible because new versions of the .NET API will be backward compatible.

> Note: It is highly recommended NOT to bind to a specific version of the AppServer assembly because you would need to upgrade your application each time a new PEAXACT version gets released.

```
using Microsoft.Win32;
using S_PACT.PEAXACT;
using System;
using System.IO;
using System.Linq;
using System.Reflection;

namespace Examples
{
    class DynamicBindingExample
    {
        static DynamicBindingExample()
        {
            // Add assembly resolver within a static constructor.
            AppDomain.CurrentDomain.AssemblyResolve +=
                new ResolveEventHandler(MyResolveEventHandler);
        }

        static void Main(string[] args)
        {
            // Create an Analyzer, resolve assembly if necessary.
            Analyzer analyzer = new Analyzer();
        }

        private static Assembly MyResolveEventHandler(object s, ResolveEventArgs e)
        {
            // Return if assembly has been loaded already.
            Assembly assembly = AppDomain.CurrentDomain.GetAssemblies()
                .FirstOrDefault(x => x.FullName.StartsWith(e.Name));
            if (assembly != null) return assembly;

            // Return if not asking for PeaxactAppServer.
            if (!e.Name.StartsWith("PeaxactAppServer")) return null;

            // Search registry for highest installed version.
            RegistryKey key = RegistryKey.OpenBaseKey(RegistryHive.LocalMachine,
                RegistryView.Registry64).OpenSubKey(@"SOFTWARE\S-PACT");
            Version maxVersion = new Version(0, 0);
            string candidatePath = "", validPath = "";
            foreach (string keyName in key.GetSubKeyNames()
                .Where(x => x.StartsWith("PEAXACT AppServer")))
            {
                try
                {
                    if (!Version.TryParse((string)key.OpenSubKey(keyName)
                        ?.GetValue("Version"), out Version version)) continue;
                    if (version <= maxVersion) continue;
                    maxVersion = version;
```

```
                    candidatePath = Path.Combine((string)key.OpenSubKey(keyName)
                        ?.GetValue("InstallPath"), "PeaxactAppServer.dll");
                }
                catch { }
                if (File.Exists(candidatePath)) validPath = candidatePath;
            }

            // Load assembly.
            if (validPath != "") return Assembly.LoadFrom(validPath);
            return null;
        }
    }
}
```

3 CUSTOM INTERFACES

3.1 OPUS Process

3.1.1 Prerequisites

Software Requirements

- OPUS 6.5 or higher
- package PROCESS for OPUS

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 will not be found be OPUS anymore.

Note: 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 the setup of an OPUS PROCESS scenario and does nothing so far. The file is located at `<INSTALLPATH>\AppServer\COM\OPUS`.

3.1.2 OPUS Configuration

3) Run the diagnosis program first to test whether the PEAXACT AppServer is installed and registered correctly.
4) Configure a new OPUS PROCESS scenario file (.obs) with the OPUS scenario browser.
5) Each measurement point requires a "No Evaluation" data channel for triggering the measurement (must be the first data channel in each case).
6) Add data channels with data evaluation by script `PeaxactComponentAnalysis.obs`.
7) Modify the scenario script according to instructions in the next section.
8) Run the process script in OPUS.

3.1.3 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 prevents from copying headers and footers.

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

```
' Added by S-PACT %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Dim pxAnalyzer
' %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

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

```
' Added by S-PACT %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Set pxAnalyzer = CreateObject("PEAXACT.Analyzer.5")
pxAnalyzer.AddModel "<ModelFileName>"
' %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Customize <ModelFileName> to load your models

- Substitute <ModelFileName> with the path to a PEAXACT model file. For instance, the line would then read:

  ```
  pxAnalyzer.AddModel "C:\Models\CyclohexaneModel.pxm"
  ```
- If you want to add more models, duplicate the pxAnalyzer.AddModel line.

At the very end of the script, add:

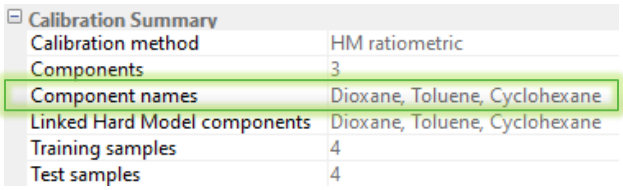
```
' Added by S-PACT %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Function PeaxactAnalysis(ByVal analysisType, ByVal block, ByVal Id)
  Dim vntResult, iPoint, nPoints, firstX, lastX, path, file, URI
  Dim xData(), yData(0), pxSample, results, result, resultValue
  vntResult = Form.OpusRequest("BINARY")
  vntResult = Form.OpusRequest("FLOAT_MODE")
  vntResult = Form.OpusRequest("FLOATCONV_MODE ON")
  vntResult = Form.OpusRequest("DATA_POINTS")
  vntResult = Form.OpusRequest("READ_FROM_BLOCK " & block)
  path = split(Form.OpusRequest("READ_PARAMETER PAT"), chr(10))(1)
  file = split(Form.OpusRequest("READ_PARAMETER NAM"), chr(10))(1)
  nPoints = split(Form.OpusRequest("READ_PARAMETER NPT"), chr(10))(1)
  firstX = split(Form.OpusRequest("READ_PARAMETER FXV"), chr(10))(1)
  lastX = split(Form.OpusRequest("READ_PARAMETER LXV"), chr(10))(1)
  ReDim xData(nPoints-1) 'Get xData
  For iPoint = 0 To nPoints-1
    xData(iPoint) = CDbl(firstX + iPoint * (lastX-firstX)/(nPoints-1))
  Next
  vntResult = Form.OpusRequestData("READ_DATA", yData) 'Get yData
  For iPoint = 0 To nPoints-1
    yData(iPoint) = CDbl(yData(iPoint+1)) 'Index shift
  Next
  ReDim Preserve yData(nPoints-1)
  URI = path & chr(92) & file & "#" & block & "-1" 'Get sample
  Set pxSample = pxAnalyzer.CreateSample(URI, xData, yData)
  Select Case UCase(analysisType) 'Perform analysis
    Case "CUSTOM","INTEGRATION","COMPONENTFITTING","PREDICTION","IDENTIFICATION"
      results = pxAnalyzer.PerformAnalysis(analysisType, (pxSample))
      Set result = FindResultById(results, Id)
      resultValue = result.Value
    Case "PREDICTIONOUTLIERIHM"
      results = pxAnalyzer.PerformAnalysis("Prediction", (pxSample))
      Set result = FindResultById(results, Id)
      Set result = FindResultByType(result.Children, 34)
      resultValue = result.Value
    Case "PREDICTIONOUTLIERPLS"
      results = pxAnalyzer.PerformAnalysis("Prediction", (pxSample))
      Set result = FindResultById(results, Id)
      Set result = FindResultByType(result.Children, 36)
      resultValue = result.Value
    Case Else : MsgBox "Invalid analysisType: " & analysisType
  End Select
  PeaxactAnalysis = vbLf & vbLf & CStr(resultValue) ' Return result
End Function
Function FindResultById(ByRef results, ByVal Id)
  Dim result
  If VarType(Id) = 8 Then ' string
    For Each result In results
      If result.Name = Id Then Set FindResultById = result : Exit Function
    Next
    Msgbox "Debugging required. Invalid result name: " & Id
  Else
    Set FindResultById = results(Id-1) : Exit Function
  End If
End Function
Function FindResultByType(ByRef results, ByVal resultType)
  Dim result
  For Each result In results
    If result.Type = resultType Then Set FindResultByType = result : Exit Function
  Next
  Msgbox "Debugging required. Invalid result type: " & resultType
End Function
' %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Search and replace the 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("<AnalysisType>", "<Block>", "<ComponentName>")
' %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

- Substitute `<AnalysisType>` with one of the following types:
  - `Custom` – calculation of custom result values.
  - `Integration` – calculation of Integration Model Component area.
  - `ComponentFitting` – calculation of Hard Model Component weight.
  - `Prediction` – calculation of Calibration Model Component value.
  - `Identification` – calculation of Classification Model Component value.
  - `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 a Hard Model; requires a HM calibration.
- Substitute `<Block>` with the desired file block, e.g., `AB`.
- Substitute `<ComponentName>` depending on your choice of `<AnalysisType>`:
  - `Custom`: substitute with the name of a custom result.
  - `Integration`: substitute with the name of an Integration Model Component.
  - `ComponentFitting`: substitute with the name of a Hard Model Component.
  - `Prediction`, `PredictionOutlierPLS`, `PredictionOutlierIHM`: substitute with the name of a Calibration Model Component. Be careful not to accidentally use names of linked components. See the model summary report:



  - `Identification`: substitute with the name of a Classification Model Component.
- 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.

3.2 HoloPro

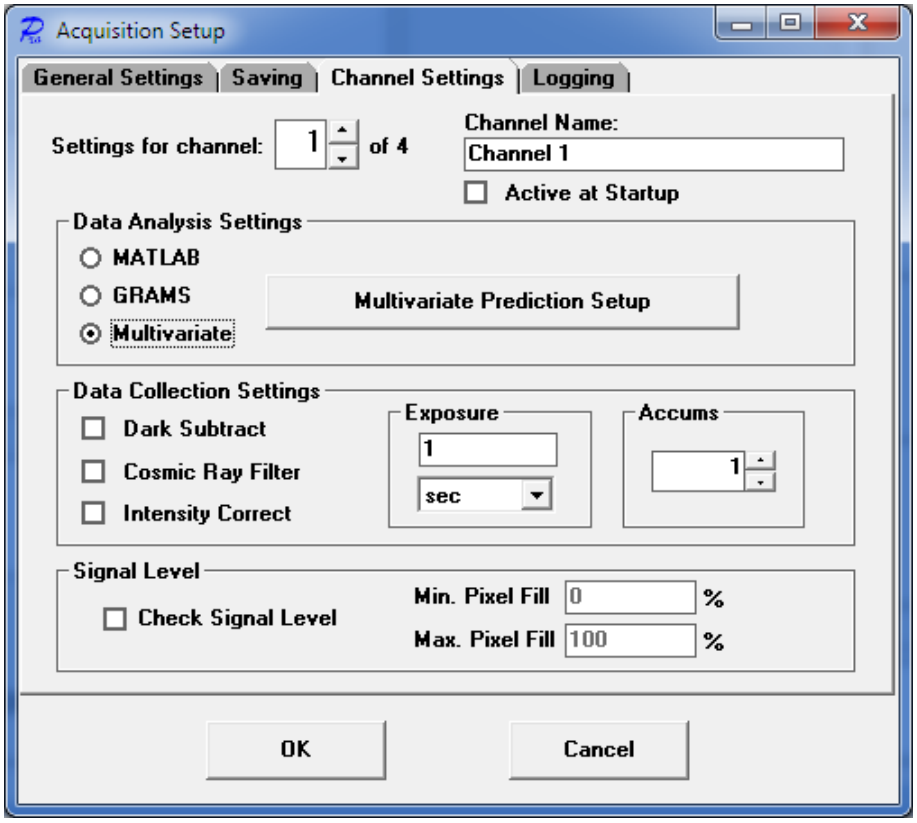
3.2.1 Prerequisites

Software Requirements

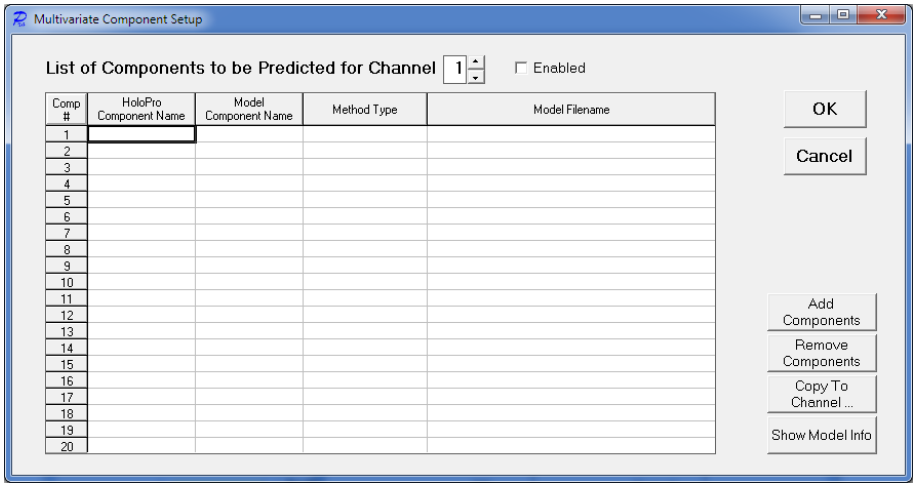
- HoloPro 3.2.0.6 to 3.3.0.3 or
- HoloPro 3.3.0.11 and newer

3.2.2 Configuration

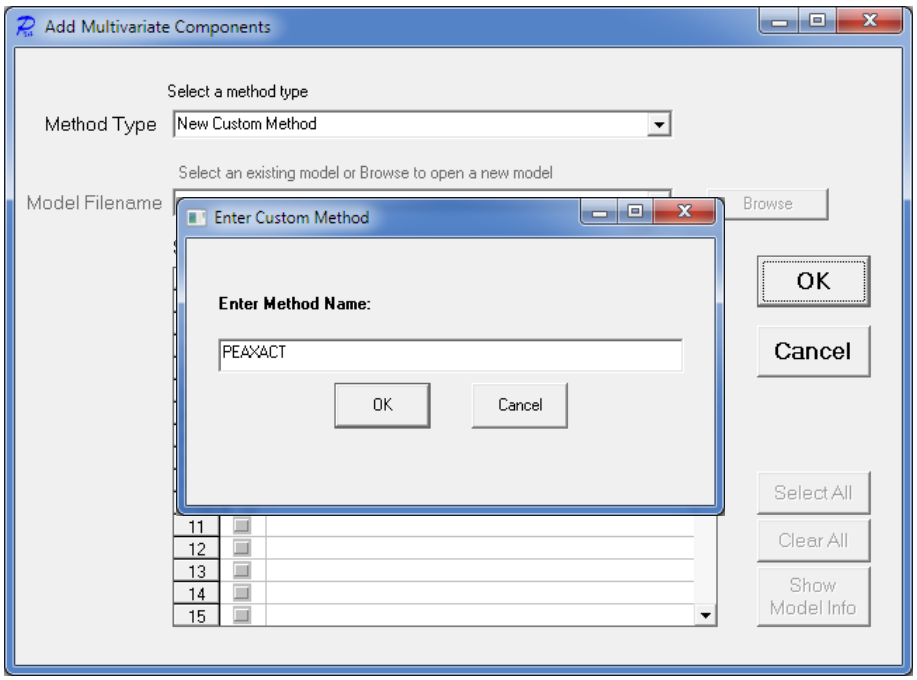
1) Run the diagnosis program first to test whether the PEAXACT AppServer is installed and registered correctly.
2) Start HoloPro and open the **Channel Settings** (menu **Settings > Acquisition Setup**)



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

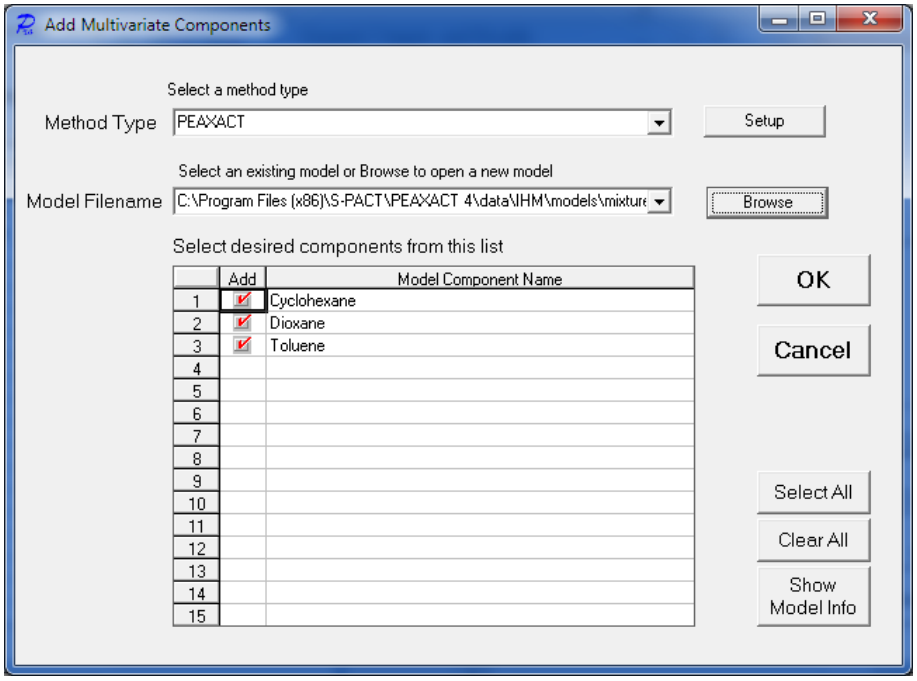


5) From the **Method Type** list select **New Custom Method** and enter `PEAXACT` (or select `PEAXACT` if it has already been added before).



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

Note: Adding the first model may take a while (because the PEAXACT AppServer gets started in the background).
Note: You can also browse for a PEAXACT session file to load multiple models from the session.



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

4 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

- Reboot the computer if you have not done this after you have installed PEAXACT.
- Run the diagnosis program. After starting the program, it performs several tests. In case of errors a possible solution is suggested. You must fix all problems before you can use the interface correctly.
- Under some circumstances the diagnosis program crashes (throwing a Windows error) when the COM DLL is registered incorrectly. If this happens, you must register the DLL manually by executing the file

  ```
  <INSTALLPATH>\AppServer\COM\Register.vbs
  ```

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

Problems with the HoloPro Custom Interface

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

Resolution

- Make sure you do not use HoloPro versions 3.3.0.7 to 3.3.0.10. These versions are known to cause problems with PEAXACT. You could use an earlier version, e.g., 3.3.0.3 or a later version, e.g., 3.3.0.11.
- See also Problems with the COM API