TechnicalNOTE

User Defined Components for EIS Modeling

Introduction

With Gamry's Echem Analyst and Model Editor you can create all sorts of EIS models provided they are made up of series and parallel combinations of 8 different basic circuit components. We've included normal electronic elements like resistors, capacitors, inductors, and other components that represent electrochemical processes, e.g. Constant Phase Element and Warburg (diffusional) Impedance.

Some users find that the 8 components are not adequate for modeling their impedance data. To handle novel circuit elements, we've come up with a way for you to create your own circuit elements. This is done by creating a dynamic link library (DLL) that contains all the information necessary to describe your new component, even including the bitmap to draw on the button or component in the Model Editor.

This DLL is written in C and compiled using Microsoft Visual C++ 2005 or 2008[1]. Each component library may contain several different component types. More than one component library can be loaded into Echem Analyst[2]. This allows you to share component libraries with other users.

Using the Echem Analyst to Fit EIS Data

Electrochemical Impedance Spectroscopy (EIS) is a very powerful tool that has been used in a wide range of applications (corrosion, energy storage devices, and organic photovoltaics, to name a few). An impedance analysis begins with measuring your cell's impedance at a series of frequencies. This is known as a spectrum. Figure 1 shows an EIS Spectrum in the Echem Analyst taken with Gamry's Universal Dummy Cell, EIS section.

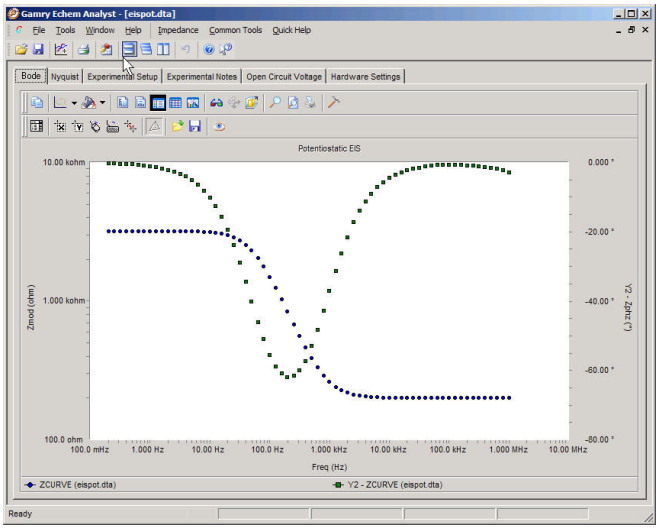


Figure 1. EIS Data (Z, Phase vs. Freq) in the Gamry Echem Analyst

After the spectrum is obtained, of course, the question becomes, "What the heck does it mean? " To answer this we use EIS modeling. An EIS **model** is an electronic circuit model which behaves, at least with respect to small sine wave signals, like the electrochemical system we are measuring. Figure 2 shows a simple model of an electrode called the "Simplified Randles' Cell" which for simplicity, we'll call the "Randles Cell". It's shown in Gamry's Impedance Model Editor. There are three **components** in this cell. Each component here has one **parameter** value.

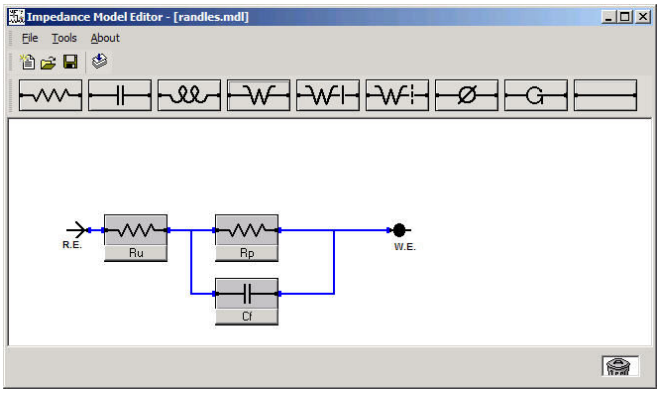


Figure 2. A Randles Cell Model

[1] Copyright Microsoft Corporation
[2] Copyright Gamry Instruments, Inc.

The words Model, Component, and Parameter above are highlighted because they show up repeatedly throughout this note. They are defined as follows:

Model: A group of circuit elements connected in series and parallel, ending with two terminals. The impedance of the whole model may be calculated.

Component: A single circuit element with two terminals. The term **Element,** e.g. Constant Phase Element, is synonymous with component.

Parameter: One or more named values that controls the impedance of a component.

In a Randles Cell, the electrode surface is shown as Faradaic capacitance, Cf, in parallel with a polarization resistance, Rp. The solution resistance, Ru, is in series with the Rp & Cf pair. You can use Echem Analyst to find a set of values for Ru, Rp, and Cf that most closely fits the experimental data (see Figure 3). Note that these terms both describe the components and their parameters.

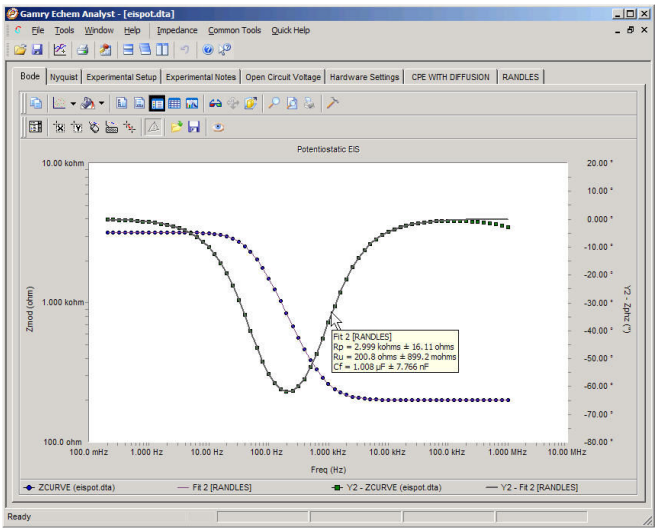


Figure 3. Best Fit With the Randle's Model

You can see this model fits our experimental data very nicely. Of course, this should be expected since the data came from a cell that is designed to look like a Randles cell circuit. At high frequencies where cable effects are starting to appear, the fit is worse.

What's in A Component?

Let's look more deeply at what makes up a component. Try this: Open the Model Editor from the Echem Analyst and start with a new blank model. Let's put a Constant Phase Element into the model (figure 4). You can display this component to show by selecting the CPE button in the model editor. It looks like a circle with a slash drawn through it.

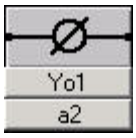


Figure 4. Constant Phase Element and Parameter Dialogs

Obviously, a component is a mathematical analog for 2 terminal electronic device, something you could measure with an impedance or ohm meter.

The CPE component has a name, "Constant Phase Element" that shows up in a tool tip. It also has a bitmap image, the Circle/Slash in this case. Finally, it has two parameters, "Yo" and "a". These parameters have index values appended to them, e.g. Yo1, a2. If another CPE is added to a model, its parameters would have different indices, e.g. Yo3 & a4, so that all parameter names are guaranteed unique.

To get to the parameter definition dialogs (see Figure 5), just click the name of the parameter. You can see each parameter has a name field that can be modified, a default initial value, and upper and lower limits. You can separately enable or disable these limits.

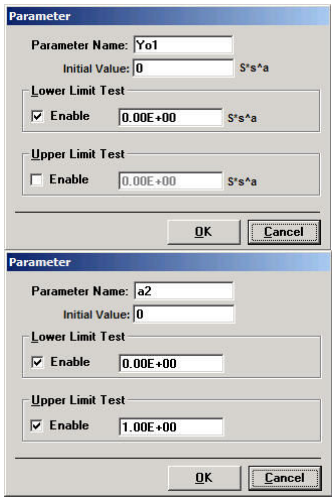


Figure 5. Parameter Default Value Dialogs

Finally, the component needs a mathematical function that specifies the component's impedance and derivatives:

$$Z(f, Yo1, a2)$$

$$\frac{\partial Z(f, Yo1, a2)}{\partial Yo1}$$

$$\frac{\partial Z(f, Yo1, a2)}{\partial a2}$$

The rest of this note describes how to create these components and load them into the Echem Analyst and Model Editor. But first, let's take a little side trip into the math of impedance modeling.

How Does Modeling Work?

To come up with the best fitting set of model parameters, the Echem Analyst EIS fitting algorithms will try to minimize a function, $\chi^2\left(\{Param_i\}\right)$, which measures the difference between the measured and model curve:

$$\chi^2\left(\{Param_i\}\right)=\sum_k\left(Z_{meas}(f_k)-Z_{model}\left(f_k,\{Param_i\}\right)\right)^2,$$

where $\{Param_i\}$ is a set of i parameters, e.g. in the Randles Cell $\{Rp, Ru, Cf\}$. There are some other issues such as weighting and handling complex numbers left out for the sake of this discussion.

In the equation $Z_{meas}\left(f_k\right)$ is the measured value of impedance at the k^{th} frequency and $Z_{model}\left(f_k,\{Param_i\}\right)$ is the calculated value at the k^{th} frequency using some particular set of parameter values. χ^2 (Chi-Squared, rhymes with Pi-Squared) is the sum of squares of differences between measured and model impedances. Being a square, is a number greater than or equal to 0.

The Echem Analyst can use either a Levenberg-Marquardt (LM) algorithm or a Simplex algorithm to find the minimum of χ^2 given a reasonable, user-supplied, starting parameter set. The LM method is faster but requires the partial derivatives, $\frac{\partial Z\left(f,\{Param_i\}\right)}{\partial Param_i}$, for each parameter. The Simplex method is more robust and does not require the derivatives during operation but it is not as fast.

At the end of either algorithm, the result is a set of parameters where χ^2 is at a minimum. Look back at Figure 3 in the little box. You'll see a value like $Rp = 2.999\ kohms \pm 16.11\ ohms$. χ^2 is also used to give estimates for the uncertainty in each parameter. The error calculation requires the partial derivatives of Z at the minimum value of χ^2, even if that minimum was found with the Simplex algorithm.

For details on how these algorithms work see Press, W.H. *et.al.* [3].

Building a Component Library

To build a component you'll need to have a Microsoft Visual C++ 2005 or Visual C++ 2008, Express Edition or better. See http://www.microsoft/express for VC++ 2008, Express Edition or http://www.microsoft.com/express/2005 for VC++ 2005.

In Windows XP, the Echem Analyst's extended component DLL's are found in the directory:

```
C:\Documents and Settings\All Users\Application Data\Gamry Instruments\Echem Analyst\Extended Components
```

In Vista, they're in

```
C:\ProgramData\Gamry Instruments\Echem Analyst\Extended Components
```

From this point on we'll abbreviate these paths as `…\Extended Components`.

The Echem Analyst will scan this directory for any DLL files and attempt to load any the components it finds in them. If you want a particular component library omitted from the Echem Analyst, just remove it from this directory.

The source for each component library is kept in a directory below the Extended Components directory. We'll create a sample project now. For example, suppose you wanted to create "MyComponentLibrary". Please note we're using VC++ 2005 here. VC++ 2008 may vary, but hopefully not too much.

The code below is included the Analyst Version 5.5 and above. You can find it at `...\Extended Components\Gamry User Defined Component`.

We're going to be building another copy of that library now.

Open Visual C++. If it starts up with the last project currently loaded, close that project.

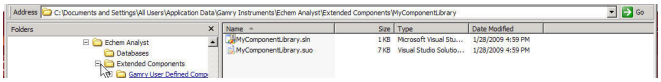
- From the menu select: File | New | Project
- From the New Project Dialog on the left side, select: Project Type = "Visual Studio Solutions".

[3] Press, W.H.; Teukolsky, S.A.; Vetterling, W.T.; Flannery, B.P. *Numerical Recipes in C, The Art of Scientific Computing, Second Edition*, Cambridge University Press,1992

- Then on the right side select: "Blank Solution".
- At the bottom enter the path:

  `...\Extended Components\`
- And enter the name: `MyComponentLibrary`

A new blank solution will be created. If you look in File Explorer, you will see something like the following:



Now let's create a project to hold the code. Note solutions and projects are different. In VC++ one can create a project but a solution will be created anyway. To create the project:

- From the menu select: File | New | Project
- From the New Project Dialog on the left side, select: Project Type = "Visual C++"
- On the right side select: Empty Project
- At the bottom set:

  Name: MyComponentLibrary

  Location: `…\Extended Components\`

  Solution: Add to Solution
- And Click: OK

A project file, MyComponentLibrary.vcproj, will be added to the solution. It would be a good idea to save all project files now.

Project Settings

There are several settings that need to be adjusted to make our DLL. To modify project settings:

- Right click on the project name, MyComponentLibrary in the Solution Explorer
- Click: Properties and a Property Pages dialog will open
- On the left pane, click: Configuration Properties | General
- On the right pane, click: Configuration Type.
- Using the drop down selector, pick: Dynamic Library (.dll)
- Click: Apply

I am going to use this shorthand to abbreviate this setting:

- Project|Property|Config|General|Configuration Type = Dynamic Library

Here some other settings I've used to create a debugable DLL

- Project|Property|Config |General|Output Directory = .. (that's 2 dots)
- Project|Property|Config |C/C++|General|Debug Information Format = Program Database
- Project|Property|Config |C/C++|Optimization|Optimization=Disabled
- Project|Property|Config |C/C++|Code Generation| Enable C++ Exceptions= No
- Project|Property|Config |C/C++|Advanced|Compile As = Default
- Project|Property|Config|Linker|Debugging| Generate Debug Info = Yes

When you want to create a release (optimized) version of DLL for best performance use:

- Project|Property|Config |C/C++|Optimization|Optimization=Maximize Speed

Adding Header Files

We're now ready to start adding code to the project. The next step is to include GamryComponent.H in your project.

- In Solution Explorer, expand MyComponentLibrary project so the subheadings are shown
- Right-Click: "Header Files"
- Click: Add | Existing Items
- Use the file finder to get to path: `…\Extended Components\include`
- Select and Add or Double Click: GamryComponent.h
- Also add an empty header file named "resource.h"
- In Solution Explorer, expand MyComponentLibrary project so the subheadings are shown
- Right-Click: "Header Files"

- Click: Add | New Item…
- In the Add New Item dialog select:

  On the right pane: Visual C++|Code

  On the left pane: Header File (.h)

  On the bottom: Name = resource.h
- Click: Add

We'll use this file for storing bitmap ID's later.

What's in GamryComponent.H

Note: You should never edit this file unless told to by a technical expert at Gamry since it matches the data structures used in the Analyst. Also, there is a version number in GamryComponent.H that must match the version of Analyst. GamryComponent.H may change in future versions so your DLL must be recompiled when Analyst is upgraded.

To create a component, there are a number of values that need to be defined. We need to know how to calculate the component impedance, whether a parameter has limits, how to display the component, and so forth. These values are contained the COMPONENT data structure and its subsequent members. The structures are all defined in GamryComponent.H.

Here is the definition of COMPONENT (as of Version 5.5):

```
typedef struct COMPONENT
        {
        char    Name[COMPONENT_NAME_LENGTH+1];

        int     ParameterCount;
        CALCZ   CalcZ;
        PARAMETER *
pParameters[MAX_PARAM_PER_COMP];
        int     ImageID;
        bool    EstimateDerivative;
        } COMPONENT;
```

where

Name: The name of the parameter, e.g. "Resistor" that will be displayed as a tooltip in the Model Editor.

ImageID: A resource ID used to retrieve the bitmap of the component. (See the section on creating and editing images.)

CalcZ: A pointer to the calculator function used to calculate Z and dZdP (see below).

ParameterCount: The number of parameters for the component.

pParameters : An array of pointers describing the parameters for this component (see below).

LM_Compatible: If True, the values of dZdP must be calculated.

Components have one or more parameters (up to MAX_PARAM_PER_COMP). Here is the structure describing a Parameter:

```
typedef struct PARAMETER
        {
        char    Name[PARAM_NAME_LENGTH+1];
        char    Unit[LABEL_LENGTH+1];
        char    UnitNorm[LABEL_LENGTH+1];
        bool    UpperLimitTest;
        double  UpperLimitValue;
        bool    LowerLimitTest;
        double  LowerLimitValue;
        }  PARAMETER;
```

Name: Default name of the parameter, e.g. "Rf". An index is added to this parameter to guarantee it is unique, e.g. "Rf1". This name may be overriden in an actual model.

Unit: The unit of the parameter, e.g. "Ohm".

UnitNorm: The per unit area-normalized, e.g. "Ohm cm2".

UpperLimitTest: A switch to indicate a default upper limit for the parameter. The value "false" (defined as 0) means no test will be performed. This may be overridden in a model.

UpperLimitValue: The value of the default upper limit. This value will be ignored if the UpperLimitTest flag is set to 0. This may be overridden in a model.

LowerLimitTest: A switch to indicate a default lower limit for the parameter. The value "false" (defined as 0) means no test will be performed. This may be overridden in a model.

LowerLimitValue: The value of the default lower limit. This value will be ignored if the LowerLimitTest flag is set to 0. This may be overridden in a model.

'UpperLimitTest' and the 'LowerLimitTest' determine whether the parameter has any default limits. Consider this definition for ResistorR:

```
static PARAMETER ResistorR = {"R","ohm",
"ohm*cm^2", false, 0.0, true, 0.0};
```

The parameter has no upper limit test (UpperLimitTest = false) and has a lower limit (LowerLimitTest = true) with a value of 0.0. This is the default limit of the component that will appear on the Model Editor. The limit can be modified either in the model editor when editing the model, or during the fitting in the EChem Analyst. Refer to figure 5 to see how these default values are displayed in the parameter dialog.

The impedance calculation function actually performs the required calculation. At a given frequency, f, and with given parameter values, P_i, the function calculates $Z(f,P_i)$ and $\frac{\partial Z(f,P_k)}{\partial P_i}$.

It takes the form:

```
void ComponentCalcZ
        (
        bool const CalculateDerivatives,
        //Input
        double const Frequency,
        //Input
        double const * const pParameter[],
        //Input
        COMPLEX * const pZ,
        //Output
        COMPLEX * const pdZdP[]
        //Output
        );
```

where

CalculateDerivatives: (Input) To save execution time, a boolean controls the derivative calculation. If "false" or 0, the derivatives are not calculated. If "true", they will be calculated.

Frequency: (Input) A double precision floating point number that represents the frequency in Hz.

pParameter[ ]: (Input) An array of pointers to the parameter values for this component.

pZ: (Output) A pointer to a COMPLEX number that will hold the calculated impedance.

pdZdP[ ]: (Output) An array of pointer to COMPLEX numbers that will hold the derivatives of the complex impedance with respect to each parameter. The indexing of this array matches the indexing of the pParameter array (i.e. the derivative with respect to the i'th Parameter is the i'th item of the pdZdP array)

As one might expect this algorithm uses complex numbers extensively. The COMPLEX data structure is made of two double precision floating numbers:

```
typedef struct COMPLEX
        {
        double Re;
        double Im;
        } COMPLEX;
```

Creating Resource.h

In the Visual Studio Solution Explorer window:

- Right Click the section Header Files
- Click: Add| New item…
- In the Add New Item Dialog:

  On the right pane: Category = Visual C++ | Code

  On the left pane: Template = Header File

  At the bottom: Name = "resource.h"

  The Location should be: "…\Extended Components\MyComponentLibrary"
- Select: Add

Here's the contents of MyComponentLibrary.h

```
#if !defined (_RESOURCE_H)
#define _RESOURCE_H

// Sample Resource ID's
#define IDB_RESISTOR          100
#define IDB_VOIGHT            101

#endif
```

Save the file.

Please note if you are using Visual Studio Standard or Professional Edition, the Resource Editor will create this file for you and has tools for creating the bitmaps.

Creating MyComponentLibrary.C

Now let's create the file "MyComponentLibrary.C".

In the Visual Studio Solution Explorer window:

- Right Click the section Source Files
- Click: Add|New Item…
- In the Add New Item Dialog:

  On the right pane: Category = Visual C++ | Code

  On the left pane: Template = C++ File

  At the bottom: Name = "MyComponentLibrary.C"

  The Location should be: "…\Extended Components\MyComponentLibrary"
- Select: Add

And a new, empty C file is added to your project. We are going to create 2 components in this library, a Resistor and a Voight Component.

Alternatively, you can copy the file GamryUserDefinedComponent.c from the sample directory and rename it to MyComponentLibrary.C.

Include Files

In MyComponentLibrary.C add:

```
#include "GamryComponent.h"

#include "MyComponentLibrary.h"
```

Resistor Calculation Function

Now lets add a component. Add the lines:

```
// Simple Resistor Element

// Parameter Definitions


static PARAMETER ResistorR = {"R","ohm",
"ohm*cm^2", false, 0.0, true, 0.0};
        //<1>

static void ResistorCalcZ (

                        bool const
CalculateDerivatives,

                        double const Frequency,

                        double const * const
pParam[],

                        COMPLEX * const pZ,

                        COMPLEX * const pdZdP[])

 {

        double Resistance = *pParam[0];

        //<2>

        COMPLEX * pdZdR = pdZdP[0];

        pZ->Re = Resistance;

        pZ->Im = 0.0;


        if (false != CalculateDerivatives)

        //<3>

        {

                pdZdR->Re = 1.0;

                pdZdR->Im = 0.0;

        }

}
```

Section <1> defines the parameter, ResistorR, i.e. the resistance of this component. It has a name ("R"), and units ("ohm") and area normalized units ("ohm*cm^2"). The last 4 parameters indicate there is no upper limit to R and a lower limit of 0.0 ohm.

The parameters used to call CalcZResistor are described above. The "const" keyword keeps the function from inadvertently modifying the input parameters or output pointers.

Line <2> declares some simplified terms, Resistance and pdZdR, for accessing parameters in the calculation. These aren't strictly necessary but keep the code readable. You could also #define terms like:

```
#define Resistance     (*pParam[0])
#define Z              (*pZ)
#define dZdR           (*pdZdP[0])
```

These would be used to simplify the code, for example:

```
Z.Re = Resistance;
Z.Im = 0.0;

…

dZdR.Re = 1.0;
dZdR.Im = 0.0;
```

This is easy to read but must be used carefully. Preprocessor errors can be difficult to debug.

Section <3> starts the derivative calculation. The Simplex Fit does not require this except to calculate error estimates. When it is used the flag, CalculateDerivatives, will be set to false except to calculate the error estimates. The Levenberg-Marquardt algorithm requires the derivative all the way through.

Voight Calculation Function

Let's look at another component, the Voigt Element. This is a parallel combination of a resistor and capacitor. However, instead of specifying the capacitance as a parameter, the time constant, τ=RC, is used. Here's the mathematical form for the Voight Element:

$$Z = \frac{R}{1 + j\omega\tau} = \frac{R}{1 + (\omega\tau)^2}(1 - j\omega\tau)$$

The rightmost equation shows Z rationalized so that it can easily be separated into real and imaginary parts.

The derivatives, also rationalized, are given by:

$$\frac{\partial Z}{\partial R} = \frac{1}{1 + \omega^2\tau^2}(1 - j\omega\tau)$$

$$\frac{\partial Z}{\partial \tau} = \frac{R\varpi}{(1 + \omega^2\tau^2)^2}\left[-2\omega\tau + j(\omega^2\tau^2 - 1)\right]$$

Here's the whole function:

```
static PARAMETER VoightRes = {"Rv","ohm",
"ohm*cm^2", false, 0.0, true, 0.0};

static PARAMETER VoightTau = {"Tau","sec",
"sec", false, 0.0, true, 0.0};

static void VoightCalcZ(
                        bool const
CalculateDerivatives,
                        double const Frequency,
                        double const * const
pParam[],
                        COMPLEX * const pZ,
                        COMPLEX * const pdZdP[])
{
        // Create simplified terms
        double Res = *pParam[0];
        double Tau = *pParam[1];
        COMPLEX * pdZdR = pdZdP[0];
        COMPLEX * pdZdT = pdZdP[1];

        double Omega = 2.0*PI*Frequency;
        double OmTau = Omega*Tau;
        double OmTauSq = OmTau * OmTau;
        double OmTauSqPlus = OmTauSq+1;
        double dRcoeff = 1.0/OmTauSqPlus;
        double Zcoeff = Res/OmTauSqPlus;
        double dTcoeff =
Res*Omega/(OmTauSqPlus*OmTauSqPlus)

        pZ->Re = Zcoeff;
        pZ->Im = Zcoeff * -OmTau;

        if (false != CalculateDerivatives)
        {
                pdZdR->Re = dRcoeff;
                pdZdR->Im = dRcoeff * -OmTau;

                pdZdT->Re = dTcoeff * -2 *
OmTau;
                pdZdT->Im = dTcoeff * (OmTauSq -
1);
        }
}
```

The Component Library Array

The last section of this file is a list of components:

```
__declspec( dllexport )
COMPONENT UserDefinedComponentLib [] =
{
        // Name, #of Parameters, Calculator
Function, ParameterArray, imageID,
EstimateDerivative
        {"Resistor", 1,
ResistorCalcZ,{&ResistorR},IDB_RESISTOR,
false},
        {"Voight",   2, VoightCalcZ,
{&VoightRes, &VoightTau},IDB_VOIGHT, false},
        {0}  //Used for stopping
};
```

This array of components lets the EIS fitting routines and Model Editor access your components. No other publicly accessible symbols are defined. It is at the bottom of the file because it refers back to several previously defined names, e.g. CalcZResistor.

The 0 at the end informs the Echem Analyst that there are no more components in this library. Do not leave it out.

Note that the names given to the CalcZ function, the Parameters, and the Bitmap ID must match the values in the COMPONENT structure.

This is the end of the file, MyComponentLibrary.C. But it isn't quite ready to compile. We have to create a bitmaps which decorate the Model Editor. That happens in the next step.

Creating the Bitmaps

Please note if you are using Visual Studio Standard or Professional Edition, the Resource Editor has tools for

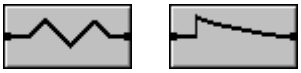
creating the bitmaps and the resource file. This section is written for users of Express Edition.

We use bitmaps to decorate the Model Editor buttons and to show the components in a model. In future versions we may use these same bitmaps to embed models into other graphics so please keep them simple.

Here's how to create one. We've put a button outline bitmap, "Blank.bmp" in the include directory. It is a 16 color, 32x64 bit bitmap which shows the button outline and terminals.

Blank.bmp

Make 2 copies of this file in your project directory, renaming them Resistor.bmp and Voight.bmp. Now you can edit each bitmap. Open each in turn using the Paint utility and decorate the bitmap as you wish. Make sure these are saved. Here's an example of the resistor & Voight bitmaps.



If you are using Visual Studio Professional edition, you may edit these bitmaps within Visual Studio. However, the Express edition does not provide graphics editing.

Adding these to the project is a bit complicated. They have to be added as imported resources. Here's the drill:

- Right Click the Resource Files section of the Project in Solution Explorer.
- Click: Add | Existing Item. An add Existing Item dialog will open.
- At the bottom select Files of type: = All Files (*.*)
- Select the three bitmap files by clicking on them while depressing the CTRL key
- Click: Add

Now do a File | Save All and you'll see the bitmap files in the Resource files list

Visual Studio will create the files "resource.h" and "MyComponentLibrary.rc".

Now add the RC file.

- Copy the file Blank.RC from the include directory to your project directory.
- Rename the copied file: "MyComponentLibrary.RC"
- Add this file to your project the same way you added the bitmap files.

And add the resource ID's

- Right click on MyComponentLibrary.RC and select: "View Code"
- Add the following lines:

```
#include "MyComponentLibrary.h"


IDB_RESISTOR BITMAP  "RESISTOR.BMP"
IDB_VOIGHT BITMAP  "VOIGHT.BMP"
```

Save this file and save the whole project.

Compiling the DLL

Now compile the whole project. By default Visual Studio will place the component library in

```
…\Extended
Components\MyComponentLibrary\Debug\MyComponent
Library.dll.
```

You can also compile this in "release" form which is more computationally efficient but harder to debug. The release version is found in

```
…\Extended Components\MyComponentLibrary\Releas
e\MyComponentLibrary.dll
```

Copying the DLL

You're now ready to copy the DLL to the directory where the Echem Analyst will be able to find it. The correct version, Debug or Release, must be copied to:

```
…\Extended Components\MyComponentLibrary.dll
```

Running the DLL

You can most easily see that the dll has loaded by running the Model Editor. Launch it from Echem Analyst or directly by running:

```
C:\Program Files\Gamry Instruments\Echem
Analyst\modelvb.exe
```

You should see the newly defined elements.

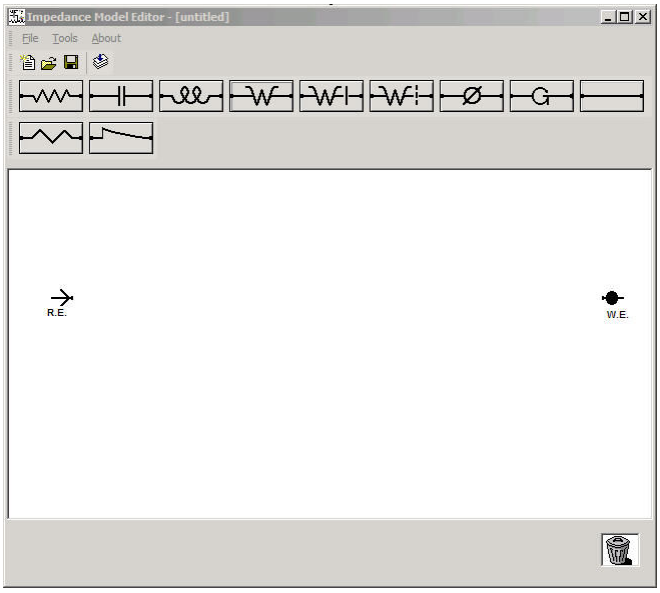
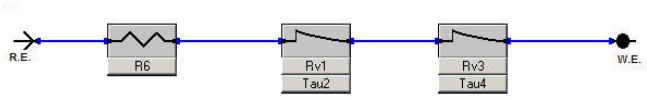


Figure 6. Model Editor Redux

Create a model, Voigt2, by putting 2 Voigt Components in series with an additional series resistor:



Now let's see if it fits. Go back to the Echem Analyst and load an EIS data file taken from your EIS dummy cell. Try the Voight fit. Here's what I get:

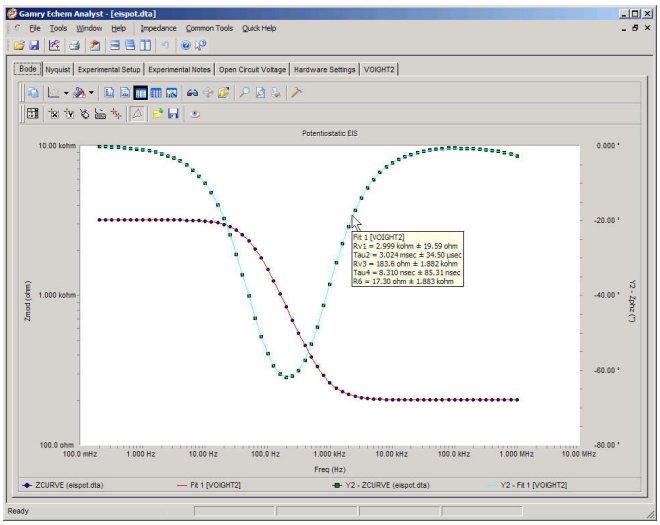


Figure 7. Voigt Fit Results

I will admit I had to sneak up on the right value of Tau4 by locking it to 1 µS, first and letting the other parameters be optimized. Then I removed the lock on Tau4 and let the Simplex find the optimum value of Tau4 = 8 nSec.

Debugging your DLL

Now just suppose your component library isn't quite correct. Or it's completely haywire. If you built the library in the Debug configuration, it is pretty simple to debug.

In the project properties set:

> Project|Property|Config |Debugging|Command =
>
> C:\Program Files\Gamry Instruments\Echem Analyst\Echem Analyst.exe

Also, right click "MyComponentLibrary" and click "Set as StartUp Project"

Then, move the cursor to the suspect line in MyComponentLibrary.c and click the F9 button. Now you can run Echem Analyst by pushing the run button  . Then open your data file normally in Echem Analyst and go through a model fit that uses the component you're trying to debug.

Debugging in Visual Studio is beyond the scope of this document but if you really get into trouble, give us a call.

Good Modeling!