GeoStudio

Add-Ins Programming Guide and Reference

GEO-SLOPE International Ltd

1400, 633 – 6th Ave SW

Calgary, Alberta, Canada T2P 2Y5

E-mail: info@geo-slope.com

Web: http://www.geo-slope.com

Table of Contents

1 Overview

GeoStudio has the ability for you to use your own "Add-In" module in place of any internal GeoStudio function or SIGMA/W constitutive model. Internal GeoStudio functions include all soil property and boundary condition functions… basically any current function that is defined as a single relationship between a "y" value and an "x" value. An example of a function internal to GeoStudio is the volumetric water content function which relates volume of water in the soil to negative pore-water pressure. Another example is the SLOPE/W function that relates shear strength to the normal effective stress at the base of a slice. An example of a boundary condition function would be a Head versus time function that reflects the rising water level in a river or lake. These examples are all based on needing to know a "y" value based on a given "x" value.

SIGMA/W has an additional use for an Add-In module. It is now possible to apply any constitutive model inside the solver. Current GeoStudio internal constitutive models include, for example, the Elastic-Plastic or Modified Cam Clay models. With an Add-In model, there is no limit to the types of stress-strain relationships that can be considered.

All functions and models set up inside GeoStudio are based on typical input parameters that the user must define. There are many cases, however, where it may be desirable to have full control over the input parameters that are used in the calculation of the "y" value of the function, or the relationship between strain and stress. For example, you may feel that your soil strength changes over time or distance. You can write an Add-In function that our solvers will use in place of the internal function. Your Add-In can access "live" data from inside our solvers or it can access data from another previously solved GeoStudio analysis or an external source such as an Excel file. It is possible to have SLOPE/W base ground strength on TEMP/W solved temperatures!

This document discusses how to make Add-Ins for GeoStudio. Once you create the Add-In module, it can be applied to any GeoStudio project. It can be shared with other users simply by e-mailing the single library file that contains the named function or model. Several functions and models can exist inside a single library file so it is possible to develop unique Add-Ins that you want your company or organization to use. The options are almost limitless.

1.1 How Does an Add-In Work?

GeoStudio Add-Ins are supplemental programs run by the solver as part of a GeoStudio analysis. Add-Ins are based on the Microsoft .NET CLR (Common Language Runtime). Any computer language compiler that can generate CLR code can be used to create an Add-In, including C#, VB.NET, and many more. A compiled CLR module that can contain GeoStudio Add-Ins is called an Assembly, and has the file extension ".dll".

An Add-In is coded as a class and the solver will generate a copy of the class at each location where the Add-In class is applied. For example, Add-Ins can be assigned to Slip Surface Slices (via strength functions), Mesh nodes (via boundary condition functions) and Mesh gauss points (via material property functions). Each copy of the class is independent but uses the same logic and main input field data as defined in the class. If the logic in the Add-In asks the solver for specific additional data, then the data used in the class is unique to the location the Add-In is applied to.

A field is a parameter variable name that will appear in, and be assigned a value in GeoStudio Define View. This allows you to create the Add-In without hard coded data such that the Add-In can be used over and over, in different models, with different input values each time. The input values are saved in the model file just as if they were part of GeoStudio.

It is important to understand that the Add-In, when applied to a model, becomes unique at each location. For example, assume a Boundary Condition Add-In Function is assigned to 3 nodes as shown below. The solver will create an Add-In object at each of the 3 nodes in memory. If the class logic asks the solver for the "x coordinate", it will be given the coordinate that belongs to the node the object is applied to. This coordinate can then be used in the calculation of the boundary condition value that should exist at that point. There are various types of data the Add-In can request from the solvers as summarized in the reference section of this document.

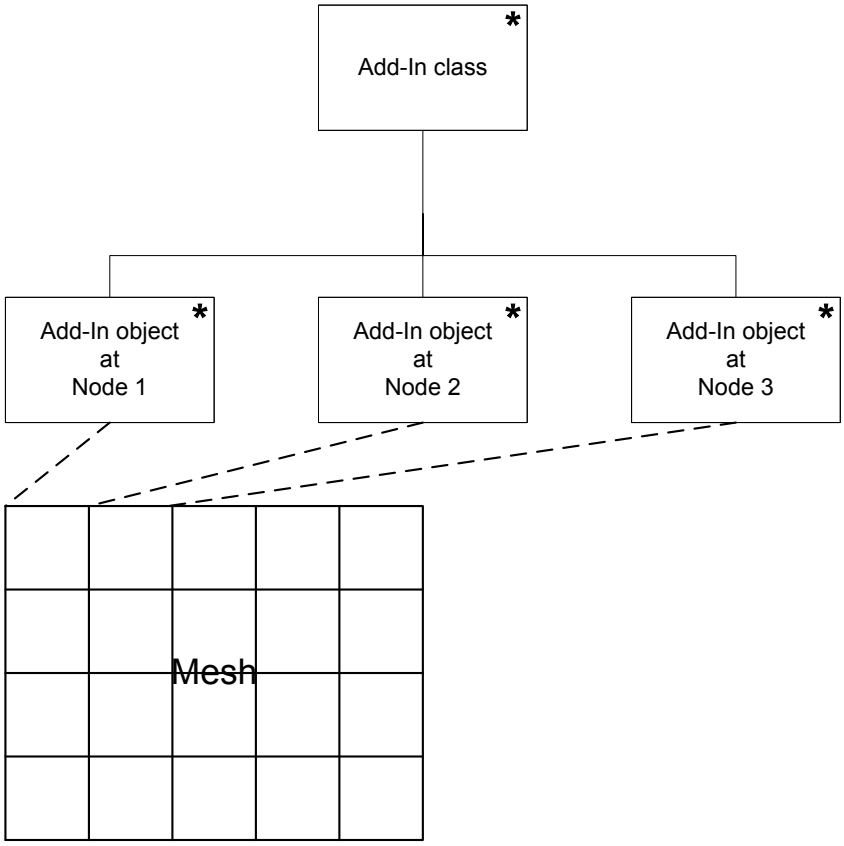


Figure 1-1 Add-In function applied to three different nodes

2 Add-In Functions (all products)

A Function Add-In is an object that takes the place of a function defined in GeoStudio, and offers the flexibility of computing function results that vary dynamically based on the current analysis state.

By default, function Add-Ins have the form y = F(x), with a single input argument and a single return value. The Add-In is passed from the solver an "x" value by default. You have the option to use the "x" value in a calculation or base the calculation or other data. Either way, the format for using a Function Add-In is that the solver will pass it an "x" value and the solver expects it to return a "y".

The data types of the "x" and "y" variables depend on the type of GeoStudio function the Add-In is replacing. For example, a conductivity function in GeoStudio is defined as Conductivity (y) versus Pressure (x). If you replace the GeoStudio function with an Add-In, then the Add-In will be passed the current pressure at a gauss point and your Add-In must return the conductivity.

The program flow is shown in the following diagram, where the solver calls the Add-In Function, which in turn requests more data from the solver about its current context before returning the function result.

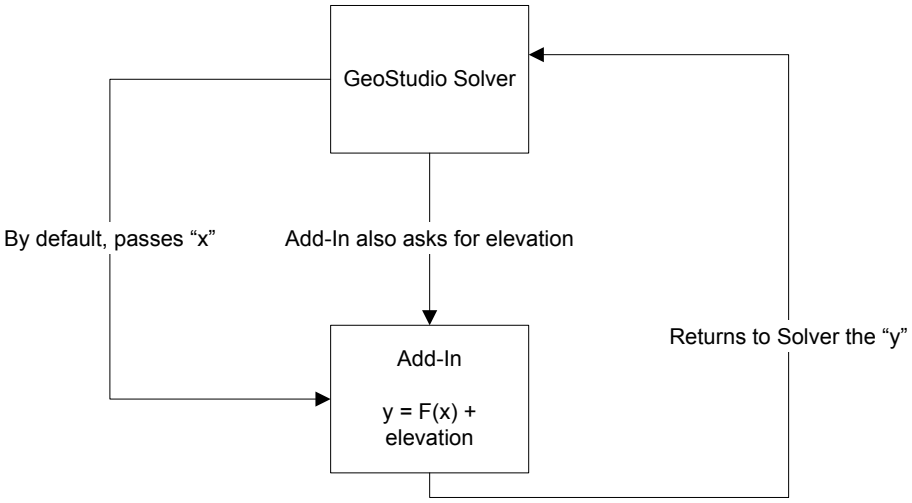


Figure 2-1 Flow logic for solver and Add-In

On supported hardware, the solver can run Add-In functions in parallel. You must be careful if your Add-In modifies shared data; either variables in your assembly or system objects like files, because many instances of the same Add-In Function could be running at the same time. As a rule of thumb, try to avoid the use of global variables/system objects in your Add-In.

2.1 Add-In Function Structure

Here is a simple Add-In Function that receives a value x from the solver and returns the same value back to the solver.

Example: Simple Add-In function

```
// A simple Add-In funciton.
// All characters after the '//' in C# are comments
public class SimpleFunction
{
    public double Calculate(double x)
    {
        return x;
    }
}
```

In this case, the class is called SimpleFunction and this will be the name that appears in GeoStudio Define View when you attempt to attach the Add-In assembly file to the model you are creating. Also note that inside the class is the actual method that does the work and that it is called Calculate(). The method MUST be called this name so that the solver can find it in the class.

Note that both the class and Calculate method must be public if they are to be seen by GeoStudio when scanning an Assembly file. The same C# code without the 'public" keyword can be compiled into an Assembly but cannot be used by GeoStudio for an Add-In Function.

2.2 Fields

A field is declared in an Add-In class as a public member variable of the class.

Example: Add-In Function with a Field

```
// An Add-In function with a field called "Scale"
public class Simple_Function_With_Field
{
    public double Scale; // "Scale" can be set in KeyIn Functions
    double Not_A_Field;  // "Not_A_Field" is not set in KeyIn Functions
    public double Calculate(double x)
    {
        return x * Scale;
    }
}
```

From the example above, both Scale and Not_A_Field are member variables of the class Simple_Function_With_Field. It is important to know that values assigned to member variables are remembered between calls to Calculate(). This means that variable values you assign data to in Define View, as well as non public variable values used by the Add-In, do not lose their status or value as the solving progresses. They can be updated as part of the logic of the Calculate() method, but they remain in memory until the solver is finished.

If the variable name is declared as public, it will show up in GeoStudio Define View as a variable you can enter data for that is used in the Calculate() method. In the GeoStudio KeyIn Functions dialog, when this example Add-In Function is referenced, the field "Scale" is displayed and can be assigned a value. When the Add-In Function object is first created in the solver, "Scale" is assigned the value entered into the KeyIn Functions dialog. The member variable "Not_A_Field" is not declared public so cannot be assigned an initial value in the KeyIn Functions dialog. Its value must be set by the code instructions in the Add-In.

2.3 Function Context

Depending on where an Add-In Function is used, there is context data inside the solver to which the Add-In has access. To get data about the Function's context, the Add-In class must inherit from "Gsi.Function" found in MNGSRV.DLL that ships with the GeoStudio product binaries.

Example: Access to solver data.

```
// Example of a function object that reads the elevation
// of the current node/gauss point/slice.
public class TestDataParamFunction : Gsi.Function
{
    public double Calculate(double pressure)
    {
        double elevation = GetParam(Gsi.DataParamType.eElevation);
        return elevation * pressure;
    }

}
```

The GetParam routine accesses data from the context associated with this function. The value selected from the Gsi.DataParamType enumeration describes the data value to return. The specific set of parameters available for a context using GetParam is documented in the References section.

2.4 Custom Parameters

A user can write and read data from Custom Parameters that are not already part of the GeoStudio data parameter list. Once you write to a Custom Parameter, you can read that data back into your Add-In and it will be saved in the solver data file for access by Results View for graphing and contouring.

Here is a line of code that sets Custom Parameter 1 to be the undrained strength, Cu.

```
SetParam(Gsi.DataParamType.eCustomParam1, Cu);
```

Here is a line that says if freezing is included, set Custom Parameter 2 to be the ground temperature.

```
if ((Freezing_On_Off == "On") ||(Freezing_On_Off == "on"))
      SetParam(Gsi.DataParamType.eCustomParam2, Temperature);
```

You may specify up to 10 custom parameters.

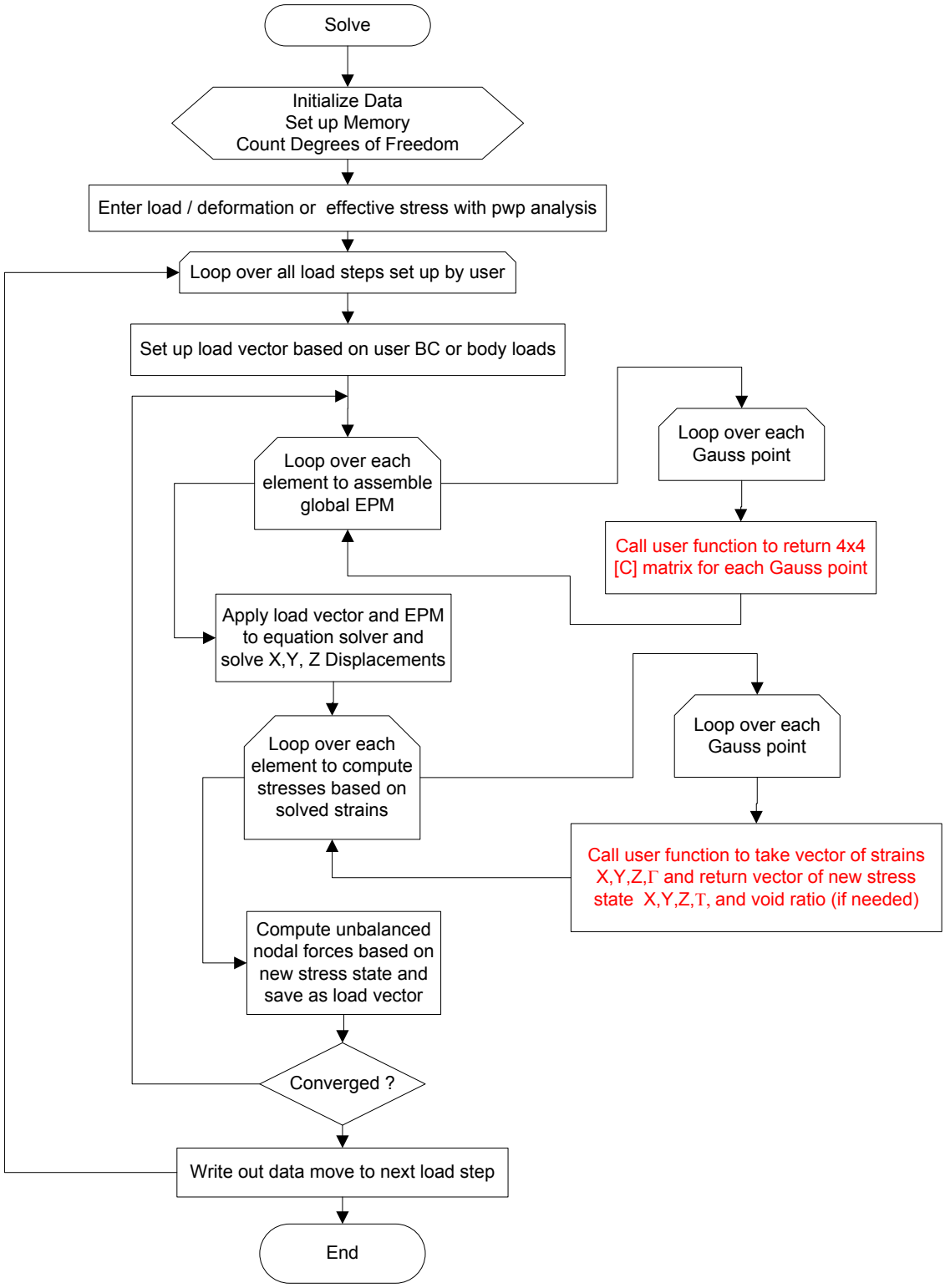
2.5 Add-In object creation

When an Add-In is created, the following sequence occurs:

1. Call the object constructor
2. Initialize all fields to the values provided.
3. If the Add-In inherits from Gsi.Function, we set the objects ID property. The ID is used by GetParam and SetParam to manipulate data unique to the Add-In's context.
4. If a public void Initialize() method exists on the object, it's called. The purpose of the Initialize() method is to perform object initialization *after* fields and the object ID have been assigned.

2.6 Add-In Constitutive Models (SIGMA/W)

Here is a basic flow chart for the SIGMA/W solver. Notice the two red colored action blocks which are the two "methods" that must exist inside a valid Add-In constitutive model.



In code form, the two necessary components of an Add-In model are highlighted in yellow below. There is a third, optional, utility method in this class that was written because the same code was needed in two places. Read the comment lines and study the code below.

```
// This is the Linear elastic model.  It has the CalculateMatrix and UpdateStresses methods so is
// a valid  Add-In constitutive model.
public class Linear_Elastic_Model : Gsi.Function
{
    // constants fields for user input
    public double E_Modulus;
    public double Poisson_Ratio;

    public void CalculateMatrix(Gsi.Matrix mCee)
    {
        CalcElasticCee(E_Modulus, Poisson_Ratio, mCee);
    }

    public void UpdateStresses(Gsi.Matrix mIncStrain, ref Gsi.Stresses mStresses)
    {
        // computed herein using Cee and Strains
        Gsi.Matrix mIncStress = Gsi.Matrix.Zero(1, 4);
        Gsi.Matrix mCee = Gsi.Matrix.Zero(4, 4);

        // compute elastic Ce used to know the stress increment
        CalcElasticCee(E_Modulus, Poisson_Ratio, mCee);

        // {IncStress} = [Ce] x {IncStrain}
        mIncStress = mIncStrain * mCee;

        mStresses.x = GetParam(Gsi.DataParamType.eXTotalStress) + mIncStress[0, 0];
        mStresses.y = GetParam(Gsi.DataParamType.eYTotalStress) + mIncStress[0, 1];
        mStresses.z = GetParam(Gsi.DataParamType.eZTotalStress) + mIncStress[0, 2];
        mStresses.xyShear = GetParam(Gsi.DataParamType.eXYShearStress) + mIncStress[0, 3];

        mStresses.voidRatio = 0.0;   //   This is an optional return parameter if it is known.
    }

    // This function is local to this code and returns a 4x4 matrix for a gauss point with elastic
    // properties only.  It takes a modulus and poisson's ratio as input.  It is used by the main
    // two functions above.
    public static void CalcElasticCee(double fE, double fPoisson, Gsi.Matrix mCee)
    {
        double fCOM = fE / ((1.0 + fPoisson) * (1.0 - 2.0 * fPoisson));
        double fCOM1 = 1.0 - fPoisson;
        mCee[0, 0] = fCOM * fCOM1;
        mCee[0, 1] = fCOM * fPoisson;
        mCee[0, 2] = fCOM * fPoisson;
        mCee[0, 3] = 0.0;

        mCee[1, 0] = fCOM * fPoisson;
        mCee[1, 1] = fCOM * fCOM1;
        mCee[1, 2] = fCOM * fPoisson;
        mCee[1, 3] = 0.0;

        mCee[2, 0] = fCOM * fPoisson;
        mCee[2, 1] = fCOM * fPoisson;
        mCee[2, 2] = fCOM * fCOM1;
        mCee[2, 3] = 0.0;

        mCee[3, 0] = 0.0;
        mCee[3, 1] = 0.0;
        mCee[3, 2] = 0.0;
        mCee[3, 3] = 0.5 * fE / (1.0 + fPoisson);
    }
}  // this is the end of the main class in this file.
```

You may have noticed in the code above that there are several instances of matrix math. There is a section later in this guide that discusses built in matrix math functions you can access in your Add-In if you link it with the MNGSRV.DLL library supplied by Geo-Slope.

3 Editing and Compiling an Add-In

There are many ways that you can edit and compile an Add-In for GeoStudio. As mentioned earlier, any .NET compatible language can be used. In the examples that follow, it is assumed that the language C-Sharp (C#) is used for the Add-In.

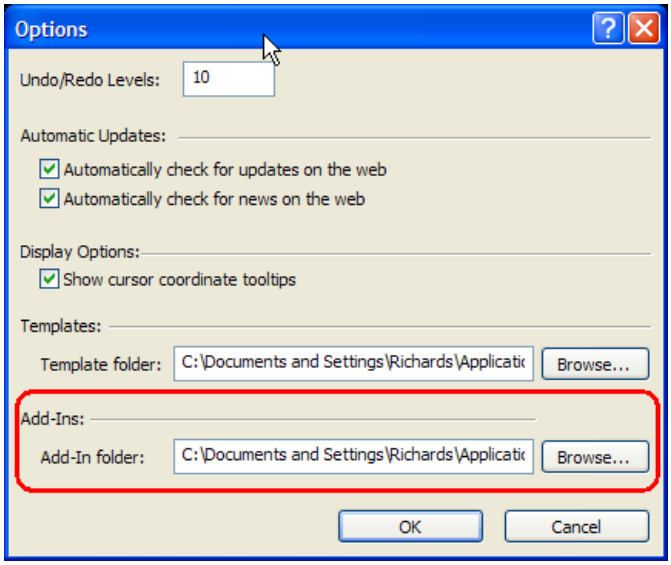
This document discusses two different ways of developing an Add-In. The first method is more suitable for simple Add-In functions where the scope of the desired function is not so large that debugging and finding errors will be difficult. This approach involves editing the code in a text editor and compiling it with a simple compiler run from Windows Explorer.

The second approach is for more complicated Add-In functions and models where there may be various supporting functions inside the same code file, or where there are many steps to the function and you need to step through each line of code to debug it and test it.

3.1 Add-In Directories

Code must be compiled into a .NET Assembly and be placed in a known Add-In directory to be used by GeoStudio.

GeoStudio scans two directories for Add-Ins: The first "AddIns" directory is located where the GeoStudio binaries are installed and is intended to be used for core Add-Ins installed as part of the GeoStudio product. The second Add-In directory is where custom Add-Ins should be placed. By default this is stored under the current users "Application Data" directory, but can be set to any directory specified by Tools – Options in GeoStudio.



3.2 Using Microsoft .NET C# compiler and a text editor

Provided with every installation of Microsoft's .NET runtime is a C# compiler: csc.exe. No other packages or programs are required to develop a GeoStudio Add-In. You can write the code in Notepad (or another text editor), save it with a *.cs extension, and then compile it using the instructions in this section.

If you write an Add-In using Notepad, you should copy the following code between the ***** lines as a starting template. Paste it into the text file and save it with a .cs file extension. You can then edit the variable names, change the name of the class to your own function name etc.

****

```
Using System;

// Example of a function object that reads the elevation
// of the current node/gauss point/slice.
public class TestDataParamFunction : Gsi.Function
{
    public double Calculate(double pressure)
    {
        double elevation = GetParam(Gsi.DataParamType.eElevation);
        return elevation * pressure;
    }

}
```

****

The C# compiler is a command line tool, and so may be a bit awkward to use. The GeoStudio Add-In developer's kit comes with a small wrapper program that runs the C# compiler with the appropriate switches and settings for creating an Add-In assembly from a C# file.

Launch gsaddin.cmd from Windows Explorer by double clicking the file name "gsaddin". Then, follow the prompt and enter the name of your c# (*.cs) text file.

If the file compiles into a usable assembly, you will see some text like that shown below.

```
AddIn Name:"Common\Tutorial AddIn.cs"
Microsoft (R) Visual C# 2005 Compiler version 8.00.50727.42
for Microsoft (R) Windows (R) 2005 Framework version 2.0.50727
Copyright (C) Microsoft Corporation 2001-2005. All rights reserved.

Press Return to Continue
```

If there is an error in an Add-In, the C# compiler will return an error message with information about the line and character where the compilation failed.

```
AddIn Name:"Common\Tutorial AddIn.cs"
Microsoft (R) Visual C# 2005 Compiler version 8.00.50727.42
for Microsoft (R) Windows (R) 2005 Framework version 2.0.50727
Copyright (C) Microsoft Corporation 2001-2005. All rights reserved.

Common\Tutorial AddIn.cs(49,34): error CS1002: ; expected

Press Return to Continue
```

The error message gives the line (49) and character column (34) in the file "Samples\Simple AddIn.cs". A short message about the error is also provided.

For more complicated functions and models, a more advanced development environment is provided by Microsoft Visual Studio 2010. A free edition is available for download from Microsoft. For more details see the section "Creating an Add-In using Visual C# 2010 Express Edition".

Once the Add-In Assembly has been created, you can reference and configure the Add-Ins from GeoStudio as discussed subsequently.

3.3 Using Visual C# 2010 Express Edition

This section provides a step by step guide to developing a GeoStudio Add-In using Microsoft's Visual C# 2010 Express Edition. A similar sequence of steps can be used when developing Add-Ins with any of the Microsoft Developer Studio products.

Any development environment that can generate .NET CLR code can be used to create a GeoStudio Add-In, however Microsoft offers "express" editions of their development tools for free download. While the express editions have less functionality than the full versions, they are more than sufficient for developing GeoStudio Add-Ins.

To Simplify Add-In development, the GeoStudio Add-In Developers Kits comes with a project template that will automatically setup your C# build environment to generate a GeoStudio Add-In Assembly. To make this template available when a New Project is created, copy it into the Developer Studio Project templates directory, usually "My Documents\Visual Studio 2010\Templates\ProjectTemplates\Visual C#". Alternatively, the "Install" program provided with the developer kit will do this for you.

If not using the project template from the Add-In Developers Kit, you can perform the same process manually using the following steps:

3.3.1 Create a New "Class Library" Project

Open the developer studio and use the File, New command to create a new "class library". Name it and save it to a drive folder.

3.3.2 Add a Reference to MNGSRV.DLL

Using the solution explorer, add a reference to MNGSRV.DLL. By default, this is found on your computer at "C:\Program Files\GEO-SLOPE\GeoStudio 8\bin\mngsrv.dll"or "C:\Program Files (x86)\GEO-SLOPE\GeoStudio 8\bin\mngsrv.dll" for 64 bit windows.

In the "Class View" for MNGSRV.dll, you can see the API exported for Add-In development under the Gsi namespace.

3.3.3 Set the assembly type (Required on XP x64 systems)

You may need to explicitly set to build an x86 assembly. Under Windows XP x64 Edition, the default is to build an x64 assembly which will cause a "BadImageFormatException" error when used with GeoStudio.

By default, Visual Studio Express does not display the platform options in the configuration manager. To display it, under "Tools – Options", Set the checkbox in the lower left corner for "Show all settings". In the same dialog, select "Projects and Solutions" from the tree view. Check "Show Advanced build configurations". Close the "Tools – Options" dialog.

Now view the Configuration Manager. The “Active solution platform” combo box should now be visible. Select “<New...>”. Under “Type or select the new platform” combo box pick “x86”. Under the “Copy settings from:” combo box select “<Empty>”.

The build settings should now be set to “x86” which will run with GeoStudio.

3.3.4 Set the Output Path

Make sure the assembly is saved in the GeoStudio Add-Ins directory. This can be modified under the Project Properties, Build tab, “Output Path.”

3.3.5 Set the Target Framework

The application Target framework must be set to the .NET Framework 4. The setting for the Target Framework is located under the Application Tab of the Project | Properties.

3.3.6 Summary

With the above steps completed, you can copy / paste some example code from this document to get you started or you can find more examples on the Geo-Slope web site. Once you have the code written, you can use the Build command in the developer studio to build the solution, compile and link the various files, and create your assembly file in the specified output directory.

You are then ready to use the Add-In as discussed in the following section.

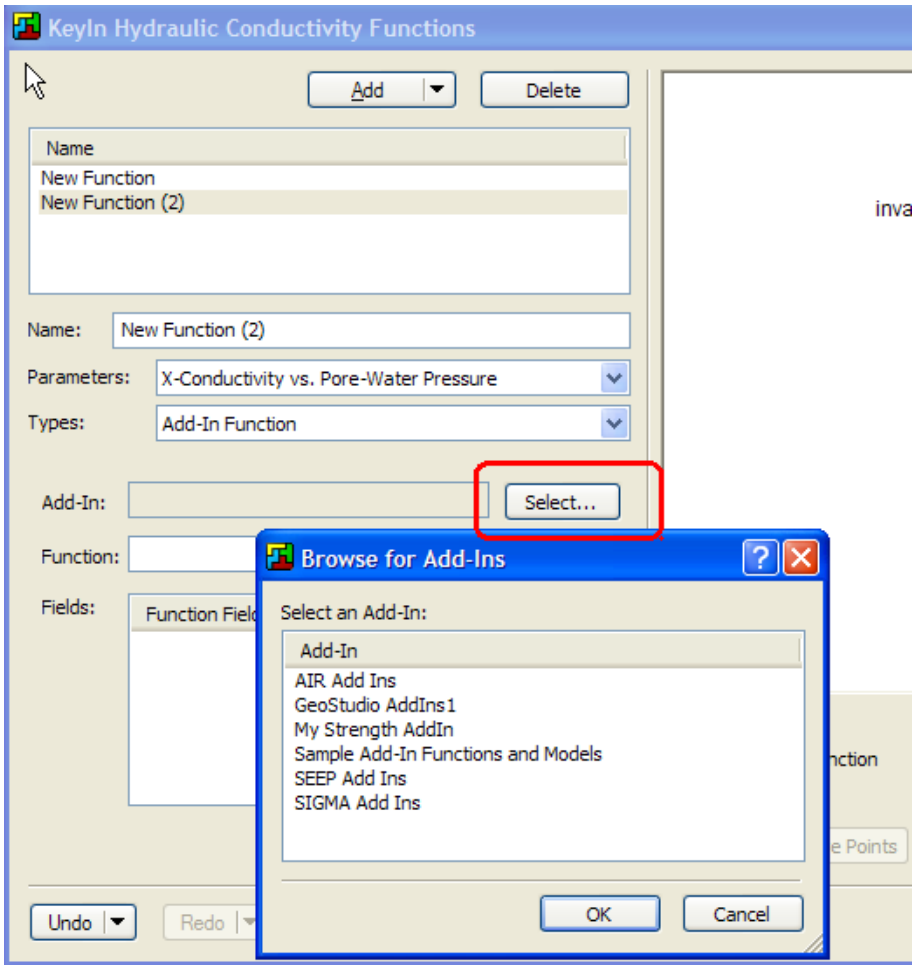
4 Referencing Add-Ins from an Analysis

The following example shows an Add-In function that will result in a sine curve function. It can be used as a boundary condition in many different analyses. The example is used to show how fields can be defined, how the function is referenced, and how the fields will appear in GeoStudio Define View.

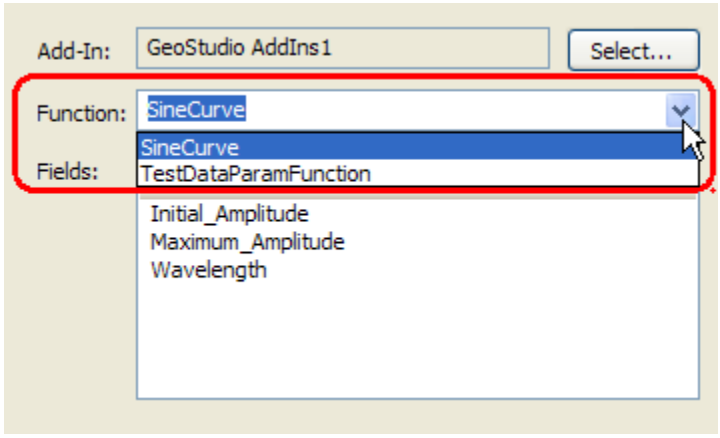
```
// This sample function takes the x value of the function type and
// returns the sine of x. Initial_Amplitude, Maximum_Amplitude
// and Wavelength are all fields that can be assigned when the
// function is selected for an analysis.

public class SineCurve
{
   // Here is the list of variables you want to enter using the
   // KeyIn Function dialog. All public variables are seen as
   // fields in the KeyIn Function dialog.
   public double Initial_Amplitude;
   public double Maximum_Amplitude;
   public double Wavelength;

   // The Calculate method is called by the SOLVER
   public double Calculate( double x )
   {
         // declare the variable to return to the solver
         double y,k,d;

         k = 6.28318531 / Wavelength;
         // phase shift
         d = Initial_Amplitude / Maximum_Amplitude;

         // calculate the value to return
         y = Math.Sin(k*x+d);

         y = y * Maximum_Amplitude;

         // return the function Y value
         return(y);
   }
}
```

4.1.1 Select the Add-In to be used for a function

In the KeyIn Functions dialog, select Add-In Function as the function Type, then press the Select button to pick from a list of Add-In assemblies found in the GeoStudio Add-Ins directories.

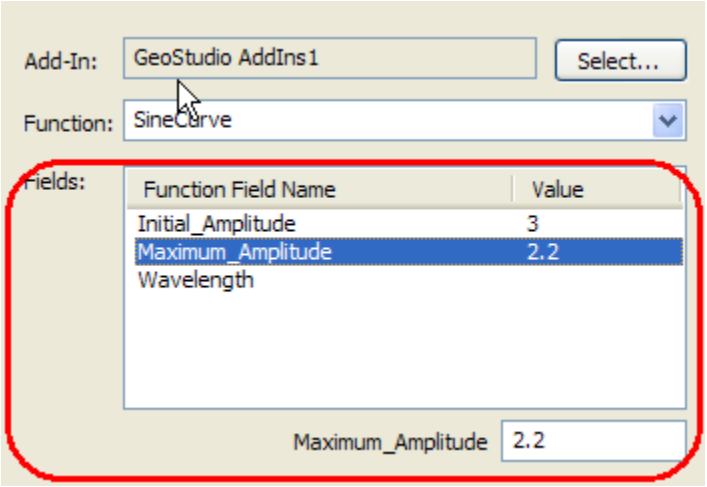


GeoStudio automatically scans the selected Assembly and looks for classes that can be used for a GeoStudio function. All valid GeoStudio Add-In Function classes must be a public and have a public method Calculate() which takes a single argument. All the Functions in the selected Add-In Assembly meeting these criteria are available to be selected using the Add-In Function combo box.



4.1.2 *Input Add-In function fields*

After selecting an Add-In function in the assembly, values for function fields can be assigned. Every instance of an Add-In function created by the solver is first initialized with the field values given.

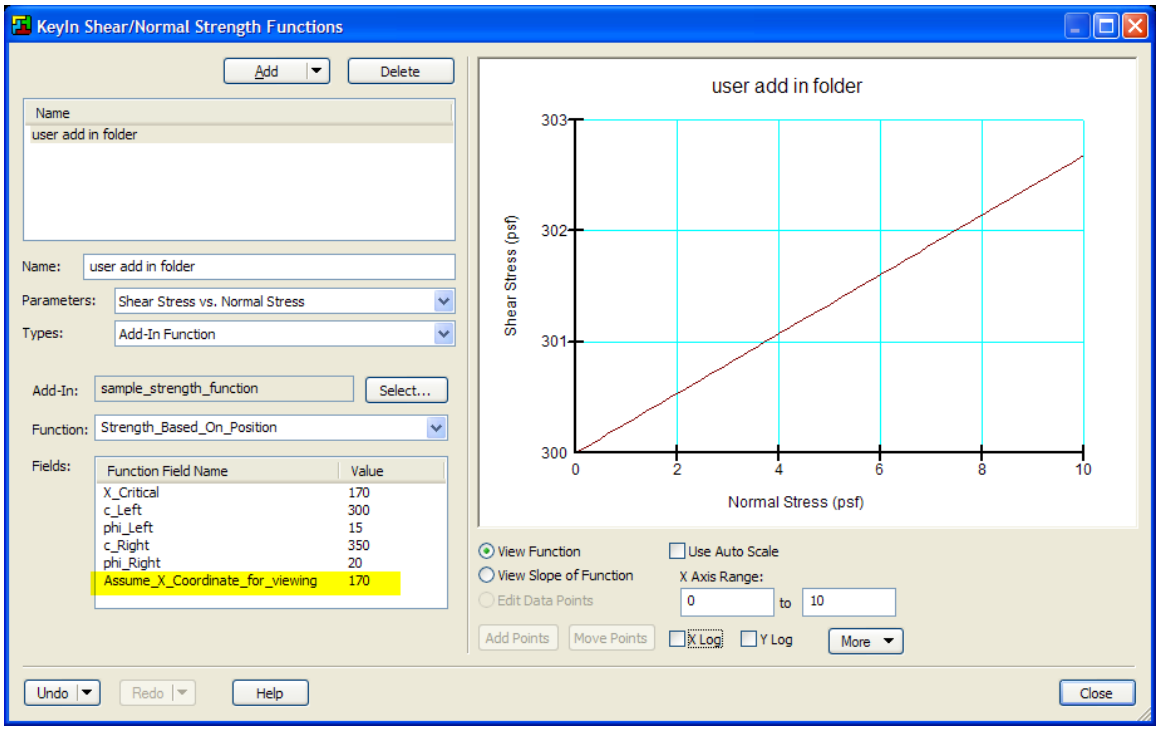


If the Add-In has valid data in it then it can be viewed in the define graph view. In some cases where the Add-In requests data from the solver, the define graph view may not have enough data to complete the calculation and present a graph. One "trick" you can use to see the function in the graph is to ask the user to enter data that the solver would normally supply. That way, the complete calculations can be made and the function can be viewed over the requested range in Define View.

The following is an example of an extra input value being requested so that the graph has valid data for display purposes. In this case, when the function is used in the solver, the x-coordinate of the slice is passed to the Add-In using a GetParam() call. However, in Define View, there is no x-value. In order to see the graph, the Add-In has an extra variable field called Assume_X_Coordinate_for_viewing that the user can enter so that the function can be fully calculated and displayed.

Here is some C# code that can be used to test if there is valid data for a calculation. If the first phrase fails, the code in the second phrase will execute.

```
// this "try" and "catch" phrase is a way to check if you have valid data.
// If not, you don't want the function to fail with an error so you can specify what to do
// if no valid data is found.

        try
        {
                BaseCenterX = GetParam(Gsi.DataParamType.eXCoord);
                // this will get live data from the solver, in this case, the x coordinate of the slice.
        }
        catch
        {
                BaseCenterX = Assume_X_Coordinate_for_viewing;
                // if GetParam() is not valid, use the Assume_X_Coordinate_for_viewing
                // so that I have something in the calcs below
        }
```


KeyIn Shear/Normal Strength Functions
Add
Delete
Name
user add in folder
Name:
user add in folder
Parameters:
Shear Stress vs. Normal Stress
Types:
Add-In Function
Add-In:
sample_strength_function
Select...
Function:
Strength_Based_On_Position
Fields:
Function Field Name
Value
X_Critical
170
c_Left
300
phi_Left
15
c_Right
350
phi_Right
20
Assume_X_Coordinate_for_viewing
170
user add in folder
Shear Stress (psf)
303
302
301
300
0
2
4
6
8
10
Normal Stress (psf)
View Function
View Slope of Function
Edit Data Points
Use Auto Scale
X Axis Range:
0
to
10
Add Points
Move Points
X Log
Y Log
More
Undo
Redo
Help
Close

5 Debugging Add-Ins

A debugger can be used to step through execution of your Add-In function while running in the solver. The simplest way is to have your debugger run solveserver.exe in the GeoStudio bin folder. The usage is given as:

<SolveServer.exe> <gsz file path> <analysis name>

The solve server can be started as an external program if you are using an integrated development environment (IDE) such as Microsoft Visual Studio. The Start Action for debugging is located under the Debug tab of the Project | Properties. For example, the start action would be set to the file location of the SolveServer:

Start external program: C:\Program Files (x86)\GEO-SLOPE\GeoStudio 8\Bin\ SolveServer.exe

and the path to the project file would be set as command line argument:

Start Options: Command line arguments: "C:\Temp\SlopeExample.gsz"

The SolveServer can be run from a command prompt with the switch /? or /help to display the command line help.

NOTE: On supported hardware, the solver can run Add-In functions in multiple threads. It's normal for the debugger to swap between the same function in different threads. To disable multiple threads of execution, set the environment variable OMP_NUM_THREADS = 1 before starting the solver at the command line.

6 Creating a Test Application

When your Add-In class library becomes complex enough, you may want to create another application to run your functions outside of GeoStudio. This allows you to easily control the input to your function and verify its results.

As long as the Add-In code does not reference the Gsi library, it can be compiled and tested separately without MNGSRV.DLL

If the Add-In uses the Gsi library to access solver data using GetParam, add the reference to MNGSRV.DLL to your test application. Before calling any routines that require Gsi library calls, run Gsi.Testing.Initialize() to setup the library.

To test a function using GetParam calls, the testing application needs to populate test values that simulate values for the Add-In context in the solver. Add-Ins use a Gsi.DataTable for GetParam calls.

Example: Populating a Gsi.DataTable with test data for GetParam calls.

```
// Initialize the Gsi library
Gsi.ExternalTesting.Initialize();
// Setup example data with two parameters.
Gsi.DataParamType[] tableParams =
{
   Gsi.DataParamType.eWaterFlux,
   Gsi.DataParamType.eVolWC
};

// Build a matrix of data with 1 row for each of the two parameters.
// Imporatant: Columns must match or an exception is thrown.
double[,] data = new double[1, 2];  //1 row by 2 columns for each table
data[0, 0] = 5.0;    //eWaterFlux   //row 0 = node id 1 in the table.
data[0, 1] = 2.0;    //eVolWC

// Wrap test data in a DataTable. The first parameter says the
// data table is indexed by node number. Use eGaussPointNum for
// functions on gauss points
Gsi.DataTable tbl =
      new Gsi.DataTable(Gsi.DataParamType.eNodeNum, tableParams, data);
// Use AssociateTable to assign a data table as the AddIn "node" table
Gsi.Testing.AssociateTable(Gsi.TableType.eNodeTable, tbl);

// Create an instance of a function to test. This function calls
// GetParam with eWaterFlux and eVolWC
MyDataParamFunction f = new MyDataParamFunction();
// Assign a function instance to the new function is calls to
// GetParam, SetParam reference the given row in the associated
// data table.
Gsi.Testing.SetFunctionInstance(f, Gsi.TableType.eNodeTable, 1);

// Start testing the function
double x = 2.0;
double y = f.Calculate(x);
```

7 MNGSRV.DLL API Reference

This section describes the API services provided by mngsrv.dll for writing an Add-In. Services for use in Add-Ins are all in the Gsi namespace. In order to use any of these services, an Add-In must reference the MNGSRV.dll assembly.

7.1 Gsi.DataTable

GeoStudio stores results in a DataTable. DataTables results from the current analysis or other analyses are available via the Gsi.DataTable object. A DataTable is a 2D array, where each column has a Gsi.DataParamType parameter name.

Gsi.DataTable's are used for writing tests and for loading results from other analyses.

7.1.1 Creation

A Gsi.DataTable can be created as a copy of a 2D array or by loading GeoStudio results from the current data file.

To create a DataTable using a 2D array of values, provide an ID type (use eNodeNum for nodal values, eGaussPointNum from gauss point values), parameter types for each column in the 2D array and the 2D array of data.

E.g.

```
// Setup example data with two parameters.
Gsi.DataParamType[] tableParams = {
   Gsi.DataParamType.eWaterFlux, Gsi.DataParamType.eVolWC
};
double[,] data = new double[1, 2];  //1 row by 2 columns for each table
data[0, 0] = 5.0;    //eWaterFlux   //row 0 = node id 1 in the table.
data[0, 1] = 2.0;    //eVolWC
Gsi.DataTable tbl =
      new Gsi.DataTable(Gsi.DataParamType.eNodeNum, tableParams, data);
```

A DataTable can be created using results from another analysis via the the Load method. Pass in the analysis name, the result type (nodal or gauss point) and the step number of the results to load.

E.g. Load nodal results from another analysis in the same project ".gsz" file.

```
Gsi.DataTable tbl = Gsi.DataTable.Load(
      sAnalysis_Name, Gsi.TableType.eNodeTable, nStep );
```

7.1.2 Get

Return a value stored in the table at the given row and parameter column.

E.g.

```
double dVal = tbl.Get( id, Gsi.DataParamType.eTemperature );
```

7.2 Gsi.Document

7.2.1 Get

Return the value of some object in the current document using its object name. Gsi.Document.Get gives access to the same data available in the GeoStudio sketch text advanced tab while in the define view.

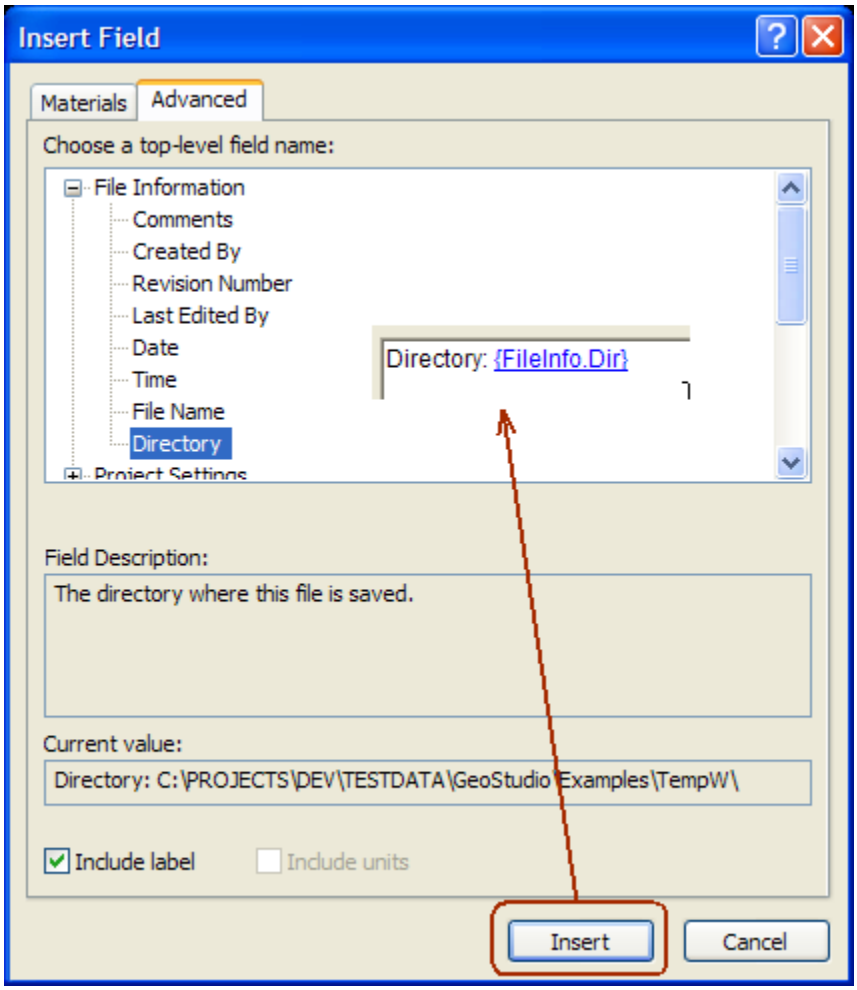
For example, to find the directory where the currently loaded GeoStudio .gsz file is found, use Gsi.Document.Get to read the value “FileInfo.Dir”:

```
String sDir = Gsi.Document.Get("FileInfo.Dir");
```

It’s possible to find the parent analysis used for initial conditions:

```
String sInitCond = Gsi.Document.Get("CurrentAnalysis.Parent");
```

The easiest way to find the names of document objects is by using the sketch text dialog in GeoStudio. Look under the Sketch – Text – Advanced tab to find a tree view of objects used in the current analysis. After using the tree view to select a document object, hit Insert to see the document object name.



7.3 Gsi.Function

An Add-In is assigned a context when its run. If the add-in is created as part of a nodal boundary condition, the Add-In function is created on every node where the boundary condition is assigned with the current node context. Similarly in SIGMA/W material modals have a current gauss point context, and in slope there is a current slice context.

The Add-In object can access data about its current context if it inherits from Gsi.Function.

See the Data Parameter Reference for information on what parameters are available for a given context and analysis type.

7.3.1 ID

This property is the context ID (i.e. node number, gauss point number or slice number).

7.3.2 GetParam

Read a data parameter for the current context.

E.g.

```
double elevation = GetParam(Gsi.DataParamType.eElevation);
```

7.3.3 SetParam

A user can write to Custom Parameters that are not already part of the GeoStudio data parameter list. Once you write to a Custom Parameter, you can read that data back into your Add-In and it will be saved in the solver data file for access by Results View for graphing and contouring.

Here is a line of code that sets Custom Parameter 1 to be the undrained strength, Cu.

```
SetParam(Gsi.DataParamType.eCustomParam1, Cu);
```

You may specify up to 10 custom parameters.

NOTE: SetParam will only work on the custom parameters; all other parameters with this context can only be directly assigned by the solver.

7.4 Gsi.Mesh

7.4.1 ComputeNodalValueAtXY

Use Gsi.Mesh.ComputeNodalValueAtXY to interpolate results from a Gsi.DataTable to some X/Y location.

See the Cohesion using Temperature example at the end of this document for an example.

7.5 Gsi.Matrix

7.5.1 Creation

The Gsi.Matrix type provides simple Matrix math services for Add-In applications.

A Gsi.Matrix can be created as a managed object with its dimension Row and Column. Note that the created matrix values are not initialized and are undefined until set.

```
// Create a 1x2 matrix
Gsi.Matrix A = new Gsi.Matrix(1,2);
A[0,0] = 1.0;
A[0,1] = 2.0;
```

You can also create a zeroed matrix using the Zero method

```
// Create a 1x2 matrix, initialize all values to 0
Gsi.Matrix A = Gsi.Matrix.Zero(1,2);
```

7.5.2 Assignment (=)

You can assign a Gsi.Matrix to another Gsi.Matrix simply by using the '=' operator. You can also assign a Gsi.Matrix to 2D array using '='.

7.5.3 Math Operators (+,-,*,/)

Most common matrix operations are performed via normal math operators. Addition, subtraction and multiplication are supported with another matrix. Scalar operations are addition, subtraction, multiplication and division.

Example: Matrix Multiply

```
// Create a 1x2 matrix
Gsi.Matrix A = new Gsi.Matrix(1,2);
A[0,0] = 1.0;
A[0,1] = 2.0;
// Create a 2x1 matrix
Gsi.Matrix B = new Gsi.Matrix(2,1);
B[0,0] = 1.0;
B[1,0] = 2.0;
B = B * 2;  //Scalar multiply
// Matrix math, create a 2x2 matrix.
Gsi.Matrix C = A * B;
```

7.5.4 Rows

Return the number of rows in the matrix such as..

```
int nRows = C.Rows;
```

7.5.5 Columns

Return the number of columns in the matrix

7.5.6 As2DArray

Convert a Gsi.Matrix into a 2D array. This method is handy to see the Matrix contents when used in the watch window of the debugger.

Example: Using As2DArray

```
// Convert a matrix into 2D C# array
double[,] data = C.As2DArray();
```

7.5.7 Transpose

Return the transpose of the passed matrix.

Example: Using Transpose

```
// mTempVCol is a 1x4 matrix, mTempVRow is a 4x1 matrix
mTempVRow = Gsi.Matrix.Transpose(mTempVCol);
```

7.5.8 Dot

Return the vector Dot product from a Gsi.Matrix. The matrix must be a row or column matrix.

Example: Vector Dot Product

```
dot = Gsi.Matrix.Dot(A, B); // dot product
```

7.5.9 TensorCross

The Tensor Cross product is supported between n x 1 and 1 x n matrices.

For the matrices a, b and c, the Tensor Cross product is C[i,j] = A[i] * B[j]

Example: Tensor Product

```
C = Gsi.Matrix.TensorCross(A, B);  // A=4x1 by B=1x4 returns 4x4
```

8 Data Parameter Reference

For every Add-In that has an assigned context, be it a node, gauss point or slip surface slice, several parameters are available using the GetParam(). For example, the following line of code obtains the elevation at gauss points or nodes:

```
double elevation = GetParam(Gsi.DataParamType.eElevation);
```

The parameter you request depends on the context of your function. Functions on nodes can access data on nodes. Functions at gauss points can access data at gauss point. This reference section discusses these options for each GeoStudio product.

Some parameters are common to all gauss points, nodes and slices. These are shown in the first table below.

Table 1 GetParam() Function Parameters Available in all Contexts

Parameter Name	Contexts	Description
eXCoord	Slices, Gauss Points, Nodes	Context object X coordinate
eYCoord	Slices, Gauss Points, Nodes	Context object Y coordinate
eZCoord	Gauss Points, Nodes	Context object Z coordinate (Finite Element Solvers only)
eElevation	Gauss Points, Nodes	If a plan view analysis, this is the Z-coordinate otherwise it's the Y-coordinate.
eTime	Gauss Points, Nodes	Elapsed time for the current iteration (Finite Element Solvers only)
eStepIterationCount	Gauss Points, Nodes	Current iteration count number (Finite Element Solvers only)
ID Note: this does not require GetParam(). It can be called as follows: int nID = ID; It is an integer value.	Gauss points, Nodes, Slices	The identification of the object

The next sections show the various parameters that are available in each product and for each type of function. A word of caution: some of the requested parameters may not be available for all analyses in a product. Also, some may only have valid data AFTER the first iteration or time step so be sure to put checks in your class logic to deal with missing data.

In the tables below there is a listing of each type of function that can be replaced by an Add-In. If the function name is ShearStress (y) versus NormalStress (x), for example, then your Add-In function will be passed the current location normal stress as a default value and the solver is expecting the Add-In to return the desired shear stress at that location. In other words, every Add-In already has one piece of data in it… that being the "x" value passed by the solver.

It is not necessary to use the GetParam() data call to access the same data passed to the Add-In as a default value.

8.1 SLOPE/W Functions

There are three different functions in SLOPE/W as summarized below.

Table 2 SLOPE/W Functions that can be Replaced with Add-Ins

Function Name	Context	Can access other data using GetParam()?
ShearStress versus NormalStress	Slices	Yes
Unit Weight versus X Coordinate	Slices	Yes (but not NormalStress or PWP)
Cohesion versus X Coordinate	Slices	Yes (but not NormalStress or PWP)
Phi versus X Coordinate	Slices	Yes (but not NormalStress or PWP)
Probability versus XCoord	None	No
ModifierFactor versus Inclination	Shear / Normal Function	No

Table 3 SLOPE/W GetParam() Function Parameters

eXCoord (midpoint of base of slice)	**Limit Equilibrium only:**
eYCoord (midpoint of base of slice)	
eXLeft	eXSeismicForce
eXRight	eYSeismicForce
eYBotLeft	eXSurchargeForce
eYBotRight	eYSurchargeForce
eYTopLeft	eXPointForce

eYTopRight	eYPointForce
eBaseLength (note the sign convention)	eXReinforcementForce
eSliceWidtheRelBaseAngle	eYReinforcementForce
eRelSurfaceAngle	eXReinforcementShearForce
eBasePhi	eYReinforcementShearForce
eYTopLayer	emAlpha
eBaseAngle	eResistingMoment
eAirPressure	eActivatingMoment
ePWP (shear normal function only)	eSliceUnitWeightWithSeismic
eSliceWeight	
eNormalEffectiveStress	**FEM method only:**
eStrengthCohesive	
eStrengthFrictional	eXTotalStress
eStrengthSuction	eYTotalStress
eResistingForce	eXYShearStress
eActivatingForce	eStabilityFactor
eSlipNum	eLiquefied

8.2 SIGMA/W Functions

There are several functions in SIGMA/W as summarized below.

Table 4 SIGMA/W Functions that can be Replaced with Add-Ins

Function Name	Context	Can access other data using GetParam()?
VolWC versus PWP	Gauss Point Value	Yes
KModifier versus YEffectiveStress	Gauss Point Value	Yes
TotalEModulus versus YTotalStress	Gauss Point Value	Yes
TotalCohesion versus YTotalStress	Gauss Point Value	Yes
Ksigma versus OverburdenPressure	Gauss Point Value	Yes
KModifier versus YEffectiveStress	Gauss Point Value	Yes
TotalEModulus versus YTotalStress	Gauss Point Value	Yes

TotalCohesion versus YTotalStress	Gauss Point Value	Yes
XDisplacement versus Time	Node Value	Yes
YDisplacement versus Time	Node Value	Yes
XBoundaryForce versus Time	Node Value	Yes
YBoundaryForce versus Time	Node Value	Yes
XBoundaryStress versus Time	Node Value	Yes
YBoundaryStress versus Time	Node Value	Yes
NormalBoundaryStress versus Time	Node Value	Yes
TanBoundaryStress versus Time	Node Value	Yes
FluidBoundaryElevation versus Time	Node Value	Yes

Table 5 SIGMA/W GetParam() Function and Constitutive Model Parameters

Parameter Name	Context	Description
eXDisplacement eYDisplacement	Node Value	The current cumulative x or y displacement computed at the end of the last load step
eXBoundaryForce eYBoundaryForce	Node Value	The computed boundary force on a displacement BC or excavated node. The data is for the last load step, not the current step.
eRotation eMoment	Beam Node Value	The computed rotation (radians) or moment on a structural beam node at the end of the last load step
eTotalHead ePWP_Excess	Node Value	The total head or excess pore-water pressure at a node computed or present at the start of the current load step. There will only be data for effective stress analysis models.
eWaterFlux eWaterCumFlux	Node Value	For Nodes with a hydraulic boundary condition in a coupled effective / pwp analysis, this is the last time step instantaneous water flux and the cumulative

		flux since the start of the analysis.
eXTotalStress eYTotalStress eZTotalStress eXYShearStress	Gauss Point Value and Default Add-In Constitutive Model Parameter	This is the initial condition stress state or computed stress state at the end of the last load step or iteration. It is updated each iteration. In an Add-In model, the previous iteration stresses are passed in to the Add-In by default and then these are updated based on new incremental strain and passed back to the solver where they are saved.
eXStrain eYStrain eZStrain eXYShearStrain	Gauss Point Value	This is the cumulative strain since the start of the analysis and computed after the incremental displacements are solved and after the new stress state is computed on each iteration.
ePWP	Gauss Point Value	This is the current pore-water pressure based on initial conditions or the solved value at the end of the last load step. It is only valid in effective stress analyses.
eVoidRatio	Gauss Point Value and Default Add-In Constitutive Model Parameter	This is the void ratio computed internally for certain soil models OR computed in an Add-In model and passed back to, and saved by, the solver at the end of the previous iteration.
eKModifier	Gauss Point Value	This is the value of the function that modifies the hydraulic conductivity based on the current vertical effective stress. It is only valid in a coupled effective stress / pwp analysis.
eXConductivity	Gauss Point Value	This is the current hydraulic conductivity at a gauss point. It is only valid in a coupled effective stress / pwp analysis.
eVolWC eVolWCSlope	Gauss Point Value	This is the current water content (by volume) and is computed using the water content function. The second value is the slope of the function. It is only valid in a coupled effective stress / pwp analysis.
eXLiqVel eYLiqVel	Gauss Point Value	This is the last time step computed x and y velocity. It is only valid in a coupled effective stress / pwp analysis.
eXGradient eYGradient	Gauss Point Value	This is the last time step computed gradients. It is only valid in a coupled effective stress / pwp analysis.

Caution: some of the requested parameters may not be available for all analyses. Also, some may only have valid data AFTER the first iteration or time step so be sure to put checks in your class logic to deal with missing data.

8.3 SEEP/W and AIR/W Functions

There are several functions in SEEP/W and AIR/W as summarized below.

Table 6 SEEP/W and AIR/W Functions that can be Replaced with Add-Ins

Function Name	Context	Can access other data using GetParam()?
VolWC versus MatricSuction	Gauss Point Value	Yes
XConductivity versus MatricSuction	Gauss Point Value	Yes
TotalHead versus Time	Node Value	Yes
TotalHead versus Volume	Node Value	Yes
WaterFlux versus Time	Node Value	Yes
WaterUnitFlux versus Time	Node Value	Yes
ModifierFactor versus PWP	Node Value	Yes
AirXConductivity versus DegSaturation	Gauss Point Value	Yes
AirPressure versus Time	Node Value	Yes

Table 7 SEEP/W GetParam() Function Parameters

Parameter Name	Context	Description
eTotalHead	Node Value	The total head at a node computed or present at the start of the current load step.
eWaterFlux eWaterCumFlux	Node Value	For Nodes with a hydraulic boundary, this is the last time step instantaneous water flux and the cumulative flux since the start of the analysis.
ePWP	Gauss Point Value	This is the current pore-water pressure based on initial conditions or the solved value at the end of the last iteration.

eXConductivity	Gauss Point Value	This is the current hydraulic conductivity at a gauss point. The Y direction value depends on the K ratio and anisotropy values specified.
eVolWC eVolWCSlope	Gauss Point Value	This is the current water content (by volume) and is computed using the water content function. The second value is the slope of the function.
eXLiqVel eYLiqVel	Gauss Point Value	This is the last time step computed x and y velocity.
eXGradient eYGradient	Gauss Point Value	This is the last time step computed gradients.
eAirXConductivity	Gauss Point Value	This is the current air conductivity at a gauss point. It is only valid in an AIR/W analysis.
eAirXVelocity eAirYVelocity	Gauss Point Value	This is the last time step computed x and y air velocity. Only valid in an AIR/W analysis.
eAirXGradient eAirYGradient	Gauss Point Value	This is the last time step computed air head gradients. Only valid in an AIR/W analysis.
eAirDensity	Gauss Point Value	The density of air at the gauss point as computed at the last iteration. Only valid in an AIR/W analysis.
eAirHead	Node Value	The total air head in an AIR/W analysis as computed at the end of the last time step.
eAirPressure	Node Value	The air pressure in an AIR/W analysis as computed at the end of the last time step.
eAirFlux eAirCumFlux	Node Values	The instantaneous and cumulative air flux at the end of the last time step. Only in an AIR/W analysis.
eAirVolFlux eAirCumVolFlux	Node Values	The instantaneous and cumulative volume air flux at the end of the last time step. Only in an AIR/W analysis.
eCumSurfWater	Surface Mesh Node Value	The total non-infiltrated or ponded volume of water sitting on a surface mesh node that has a small “q” unit flux boundary condition applied to it.
eAirContent	Gauss Point Value	The air content at the last iteration. Is equal to the porosity minus the water content. Is valid for SEEP/W or AIR/W analyses.

eTemperature	Gauss Point Value	In a TEMP/W coupled AIR/W and SEEP/W analysis, this is the last time step gauss point temperature. Its used to update the air density.

Caution: some of the requested parameters may not be available for all analyses. Also, some may only have valid data AFTER the first iteration or time step so be sure to put checks in your class logic to deal with missing data.

8.4 VADOSE/W Functions

There are several functions in VADOSE/W as summarized below.

Table 8 VADOSE/W Functions that can be Replaced with Add-Ins

Function Name	Context	Can access other data using GetParam()?
VolWC versus MatricSuction	Gauss Point Value	Yes
XConductivity versus MatricSuction	Gauss Point Value	Yes
TotalHead versus Time	Node Value	Yes
TotalHead versus Volume	Node Value	Yes
WaterFlux versus Time	Node Value	Yes
WaterUnitFlux versus Time	Node Value	Yes
ModifierFactor versus PWP	Node Value	Yes
ThermalConductivity versus VolWC	Gauss Point Value	Yes
VolHeatCapacity versus VolWC	Gauss Point Value	Yes
LeafAreaIndex versus Days	Climate BC	No
LimitingFactor versus MatricSuction	Climate BC	No
RootDepth versus Days	Climate BC	No

Temperature versus Time	Node Value	Yes
ModifierFactor versus Temperature	Node Value	Yes
GasConcentration versus Time	Node Value	Yes
GasFlux versus Time	Node Value	Yes
GasUnitFlux versus Time	Node Value	Yes
ModifierFactor versus GasConcentration	Node Value	Yes

Table 9 VADOSE/W GetParam() Function Parameters

Parameter Name	Context	Description
eTotalHead	Node Value	The total head at a node computed or present at the start of the current time step.
eWaterFlux eWaterCumFlux	Node Value	For Nodes with a hydraulic boundary, this is the last time step instantaneous water flux and the cumulative flux since the start of the analysis.
eTemperature	Node Value	The temperature at a node computed at the start of the current time step.
eGasConcentration	Node Value	The oxygen or radon gas concentration at a node computed at the start of the current time step. Will be zero or missing if the gas analysis option is off.
eGasFlux	Node Value	The computed instantaneous gas flux at a gas boundary condition node at the end of the last last time step. Will be missing if gas analysis is turned off.
ePWP	Gauss Point Value	This is the current pore-water pressure based on initial conditions or the solved value at the end of the last iteration.
eXConductivity	Gauss Point Value	This is the current hydraulic conductivity at a gauss point. The Y direction value depends on the K ratio and anisotropy values specified.

eVolWC eVolWCSlope	Gauss Point Value	This is the current water content (by volume) and is computed using the water content function. The second value is the slope of the function.
eXLiqVel eYLiqVel	Gauss Point Value	This is the last time step computed x and y velocity.
eXGradient eYGradient	Gauss Point Value	This is the last time step computed gradients.
eIceContent	Gauss Point Value	This is the volumetric ice content as of the end of the last iteration.
eVapPressure	Gauss Point Value	This is the vapor pressure in the soil as of the end of the last iteration.
eGasDiffCoeff	Gauss Point Value	This is the gas (radon or oxygen) diffusion coefficient as of the end of the last iteration.
eTemperature	Gauss Point Value	This is the temperature as of the end of the last iteration.
eUnfrozenWC	Gauss Point Value	This is the unfrozen water content as of the end of the last iteration.
eAirContent	Gauss Point Value	This is the volumetric air content as of the end of the last iteration. It is equal to the porosity minus the water content minus the ice content.
eUnfrozenWCSlope	Gauss Point Value	This is the change in unfrozen water content per change in temperature at last or current iteration, depending on when you ask for this value.
eVolHeatCapacity	Gauss Point Value	This is the volumetric heat capacity at the current or last iteration, depending on when you ask for the value.
eThermalConductivity	Gauss Point Value	This is the current thermal conductivity.
eXVapVel eYVapVel	Gauss Point Values	These are the gauss point vapor velocities as of the end of the last iteration.

Caution: some of the requested parameters may not be available for all analyses. Also, some may only have valid data AFTER the first iteration or time step so be sure to put checks in your class logic to deal with missing data.

8.5 TEMP/W Functions

There are several functions in TEMP/W as summarized below.

Table 10 TEMP/W Functions that can be Replaced with Add-Ins

Function Name	Context	Can access other data using GetParam()?
ThermalFlux versus Time	Node Value	Yes
ThermalUnitFlux versus Time	Node Value	Yes
ModifierFactor versus Temperature	Node Value	Yes
ThermalConductivity versus VolWC	Gauss Point Value	Yes
VolHeatCapacity versus VolWC	Gauss Point Value	Yes
ThermalConductivity versus Temperature	Gauss Point Value	Yes
VolHeatCapacity versus Temperature	Gauss Point Value	Yes
Temperature versus Time	Node Value	Yes
Unfrozen water content versus Temperature	Gauss Point Value	Yes

Table 11 TEMP/W GetParam() Function Parameters

Parameter Name	Context	Description
eTemperature	Node Value	The temperature at a node computed at the start of the current time step.
eThermalFlux	Node Value	The thermal flux at a node at the end of the last time step.
eThermalCumFlux	Node Value	The cumulative thermal flux at a node since the start of the analysis.

eXThermalConductivity	Gauss Point Value	This is the current thermal conductivity at a gauss point. The Y direction value depends on the K ratio and anisotropy values specified.
eXThermalUnitFlux eYThermalUnitFlux	Gauss Point Values	This is the last time step computed x and y thermal flux within an element.
eXThermalGradient eYThermalGradient	Gauss Point Values	This is the last time step computed gradients.
eTemperature	Gauss Point Value	This is the temperature as of the end of the last iteration.
eUnfrozenWC	Gauss Point Value	This is the unfrozen water content as of the end of the last iteration.
eUnfrozenWCSlope	Gauss Point Value	This is the change in unfrozen water content per change in temperature at last or current iteration, depending on when you ask for this value.
eVolHeatCapacity	Gauss Point Value	This is the volumetric heat capacity at the current or last iteration, depending on when you ask for the value.
eXLiqVel eYLiqVel	Gauss Point Values	These are the gauss point liquid velocities as of the end of the last iteration if the analysis is coupled with SEEP/W.
eAirXVelocity eAirYVelocity	Gauss Point Values	These are the gauss point air velocities as of the end of the last iteration if the analysis is coupled with SEEP/W and AIR/W.
eAirContent	Gauss Point Value	This is the volumetric air content as of the end of the last iteration. It is equal to the porosity minus the water content minus the ice content. Only valid if the analysis is coupled with SEEP/W and AIR/W.
eAirDensity	Gauss Point Value	The gauss point air density at the end of the last iteration if the analysis is coupled with SEEP/W and AIR/W.
eVolWC	Gauss Point Value	This is the current water content (by volume) and is computed using the water content function. The value is only valid if the analysis is coupled with SEEP/W.

Caution: some of the requested parameters may not be available for all analyses. Also, some may only have valid data AFTER the first iteration or time step so be sure to put checks in your class logic to deal with missing data.

8.6 CTRAN/W Functions

There are several functions in CTRAN/W as summarized below.

Table 12 CTRAN/W Functions that can be Replaced with Add-Ins

Function Name	Context	Can access other data using GetParam()?
MassFlux versus Time	Node Value	Yes
MassUnitFlux versus Time	Node Value	Yes
Concentration versus Time	Node Value	Yes
Concentration versus Mass	Node Value	Yes
Adsorption versus Concentration	Gauss Point Value	Yes
Diffusivity versus Water Content	Gauss Point Value	Yes
Source Concentration versus Time	Node Value	Yes

Table 13 CTRAN/W GetParam() Function Parameters

Parameter Name	Context	Description
eConcentration	Node Value	The concentration at a node computed at the start of the current time step.
eMassFlux	Node Value	The mass flux at a node at the end of the last time step.
eMassCumFlux	Node Value	The cumulative mass flux at a node since the start of the analysis.
eTotalHead	NodeValue	The total head from SEEP/W or VADOSE/W at the start of the iteration.

eWaterFlux	Node Value	The nodal volume water flux from SEEP/W or VADOSE/W at the start of the iteration.
eXXDispersiveCoef eYYDispersiveCoef eXYDispersiveCoef	Gauss Point Values	This is the coefficient of dispersivity in each direction at the start of the iteration.
ePWP	Gauss Point Value	This is the pore-water pressure at of the end of the last iteration as obtained from SEEP/W or VADOSE/W.
eXPeclet eYPeclet	Gauss Point Values	These are the Peclet numbers for this iteration.
eXCourant eYCourant	Gauss Point Values	These are the Courant numbers for this iteration.
eConcentration	Gauss Point Value	This is the concentration at the Gauss point for the current iteration.
eXLiqVel eYLiqVel	Gauss Point Values	These are the gauss point liquid velocities as of the end of the last iteration if the analysis is coupled with SEEP/W.
eVolWC	Gauss Point Value	This is the current water content (by volume) and is computed using the water content function. The value is only valid if the analysis is coupled with SEEP/W.
eTotalMass eFluidMass eSolidMass	Gauss Point Values	These are the mass values at a gauss point at the start of the iteration. Mass is in the fluid, and adsorbed to solids. The total mass is the sum.
eAdsorption eAdsorptionSlope	Gauss Point Values	This is the current adsorption coefficient and the slope of the adsorption function at the current iteration.

Caution: some of the requested parameters may not be available for all analyses. Also, some may only have valid data AFTER the first iteration or time step so be sure to put checks in your class logic to deal with missing data.

8.7 QUAKE/W Functions

There are several functions in QUAKE/W as summarized below. At this time, none of the QUAKE/W Add-In functions can access live internal solver data other than the “x” value passed by default to the function.

Table 14 QUAKE/W Functions that can be Replaced with Add-Ins

Function Name	Context	Can access other data using GetParam()?
Gmax versus YEffectiveStress	Gauss Point Value	No
Kalpha versus YEffectiveStress	Gauss Point Value	No
CyclicNumber versus ShearStressRatio	Gauss Point Value	No
PWPRatio versus CyclicNumberRatio	Gauss Point Value	No
GGMaxRatio versus CyclicShearStrain	None	No
EQDampingRatio versus CyclicShearStrain	None	No
EModulus versus YEffectiveStress	Gauss Point Value	No
RecModulus versus YEffectiveStress	Gauss Point Value	No
IncVolStrain versus AccVolStrain	Gauss Point Value	No
XDisplacement versus Time	Node Value	No
YDisplacement versus Time	Node Value	No
XBoundaryForce versus Time	Node Value	No
YBoundaryForce versus Time	Node Value	No

9 Sample Code

Here is some sample code you can copy and paste to get started. If you copy and paste into Notepad as unformatted text, you can then save it with a *.cs extension and use the compiler discussed in section 4 above to make the actual library file.

Some of the methods below have been highlighted in different colors and can be interpreted as follows:

Yellow: The main methods that MUST exist in the Add-In

Blue: Supporting local methods that are used within other methods for convenience

Green: The Class name that will appear as an option in Define View.

9.1 SLOPE/W Strength_Based_On_Position

This function can be applied to the Shear Strength versus Normal Stress function in SLOPE/W. In this Add-In version, the function prompts the user for a C and Phi to the left and right of an arbitrary X coordinate, also input through the function. If the actual slice coordinate during the solving is to the left of the X value, one set of C and Phi are used. If the slice X is to the right of the arbitrary X coordinate, the second set of C and Phi are used. The function also asks for an extra X value to be input for viewing purposes. This will be used in Define View for seeing the function graph because there is no data passed by the solver at the Define View level.

```
using System;

public class Strength_Based_On_Position : Gsi.Function
{
        // here is the list of variables you want to enter in DEFINE.

        public double X_Critical;
        public double c_Left;
        public double phi_Left;
        public double c_Right;
        public double phi_Right;

        // Normally the solver will pass the slice X-coordinate to the function, but
        // if you want to view the graph in Define View, you can enter an arbitrary X
        // coordinate of a slice

        public double Assume_X_Coordinate_for_viewing;

        // These variables are not public so will not show up in Define View.
        // An instance of each of these will be owned by each slice
        // the function is applied to.  It's value will be retained until the function closes.

        double Strength;

        // this is the main function calculator.  You CANNOT change the name. You
        // can change the variable name.

        // EffectiveBaseNormalStress is the passed in x value of the function type.
        // You can use it and / or get a "live" value from the solver while its solving.
```

```
public double Calculate( double EffectiveBaseNormalStress )
{

        // these are local variables used in the calculation of the value this function
        // will return.

        double BaseCenterX;
        double Degree_to_Radian =  3.141592654 / 180.0;

        // this "try" and "catch" phrase is a way to check if you have valid data.
        // If not, you don't want the function to die so you can specify what to do
        // if no valid data is found.

        try
        {
                BaseCenterX = GetParam(Gsi.DataParamType.eXCoord);
                // this will get live data from the solver,
                // in this case, the x coordinate of the slice.
        }
        catch
        {
                BaseCenterX = Assume_X_Coordinate_for_viewing;
                // if I cant get solve slice x data, use the
                // Assume_X_Coordinate_for_viewing
                // so that I have something in the calcs below when viewing the
                // function in Define View.
        }

        // check if slice x is to the left or to the right of the critical X
        if(BaseCenterX < X_Critical)
        {
                // Mohr Coulomb strength at slice base
                Strength =  c_Left +
                        EffectiveBaseNormalStress * Math.Tan( phi_Left *
                        Degree_to_Radian );
        }
        else
        {
                // Mohr Coulomb strength at slice base
                Strength =  c_Right +
                        EffectiveBaseNormalStress * Math.Tan( phi_Right *
                        Degree_to_Radian );
        }

        // return the computed Strength value
        return(Strength);

}  // this is the end of the calculate method

} // this is the end of the first Class in this file.
```

9.2 SLOPE/W Strength_Between_X_Coordinates

This function can be applied to the Shear Strength versus Normal Stress function in SLOPE/W. In this Add-In version, the function prompts the user for a C and Phi at one X coordinate and another C and Phi at a second X coordinate. The function then assumes the C and Phi vary

linearly between the two coordinates. An additional parameter is used for viewing the function in Define View.

```
using System;

public class Strength_Between_X_Coordinates : Gsi.Function
{
	public double X_Coordinate_Left;
	public double c_Left;
	public double phi_Left;

	public double X_Coordinate_Right;
	public double c_Right;
	public double phi_Right;

	// Normally the solver will pass the slice X-coordinate to the function, but
	// if you want to view the graph in Define View, you can enter an arbitrary X
	// coordinate of a slice
	public double Assume_X_Coordinate_for_viewing;

	double fCInt,fPhiInt,fCSlope,fPhiSlope;

	// this is the main function calculator. You CANNOT change the name. You can change
	// the variable name.
	public double Calculate( double EffectiveBaseNormalStress )
	{
		double BaseCenterX;
		double c_New;
		double phi_New;
		double Degree_to_Radian = 3.141592654 / 180.0;

		// read in X-coordinate at slice base center
		try
		{
			BaseCenterX = GetParam(Gsi.DataParamType.eXCoord);
			// this will get live data from the solver.
		}
		catch
		{
			BaseCenterX = Assume_X_Coordinate_for_viewing;
		}

		// compute the linear c equation
		fCSlope = (c_Right - c_Left) / (X_Coordinate_Right - X_Coordinate_Left);
		fPhiSlope = (phi_Right - phi_Left) / (X_Coordinate_Right - X_Coordinate_Left);

		fCInt = c_Left - fCSlope * X_Coordinate_Left;
		fPhiInt = phi_Left - fPhiSlope * X_Coordinate_Left;

		c_New = fCInt + fCSlope * BaseCenterX;
		phi_New = fPhiInt + fPhiSlope * BaseCenterX;

		// Mohr Coulomb strength at slice base
		double Strength = c_New + EffectiveBaseNormalStress * Math.Tan( phi_New *
				Degree_to_Radian );
```

```
            // return the function Strength value
            return(Strength);
        }

}
```

9.3 SLOPE/W Mohr Coulomb

This function is a sample of how the default "x" value of a function can be used inside the Calculate() method. Notice there is no reference to Gsi.Function as it is not necessary if no GetParam() data calls are needed.

```
using System;

public class Strength_Mohr_Coulomb
{
        public double c;
        public double phi;

        // this is the main function calculator.  You CANNOT change the name. You
        // can change the variable name.
        public double Calculate( double EffectiveBaseNormalStress )
        {
            double Degree_to_Radian =  3.141592654 / 180.0;

            // Mohr Coulomb strength at slice base
            double Strength =  c + EffectiveBaseNormalStress * Math.Tan( phi *
                                    Degree_to_Radian );

            // return the function Strength value
            return(Strength);
        }

}
```

9.4 SIGMA/W Non_Uniform_Edge_Stress_BC

This function can be used to apply a non uniform edge stress boundary condition on a line geometry object. It asks for the two x,y coordinates and two stress values. It will linearly interpolate the stress value between the two end points and apply that value to any mesh nodes along the line.

```
using System;

public class Non_Uniform_Edge_Stress_BC : Gsi.Function
{
        // stress distribution input parameters

        public double Xa, Ya, Xb, Yb, Stress_a, Stress_b;

        // declare the variable to return to the solver.
        // Its is not public so won't show up as a UI item.
        // An instance of it will be owed by each node the BC function is applied to.
        // It's value will be retained until the function closes.

        double Stress;
```

```
        // Time is the passed in "x" type of the stress function.
        // Time is not used in this version but it is here because it
        // is the default function type.
        // We call internal coordinates to pass back the stress vs position

        public double Calculate( double time )
        {

                double X;   // this will get live data from the solver.
                double Y;

                // attempt to get information from the solver
                try
                {
                        X = GetParam(Gsi.DataParamType.eXCoord);
                        Y = GetParam(Gsi.DataParamType.eYCoord);
                }
                catch
                {
                        // if the solver is not running, use the passed in default "x" value
                        // as our X value and the Y value as the users second y coordinate.
                        X = time;
                        Y = Yb;
                }

                double Slope = (Stress_b - Stress_a) / Math.Sqrt( (Yb - Ya)*(Yb - Ya)  + (Xb -
                                Xa)*(Xb - Xa) );
                Stress = Stress_a + Slope * Math.Sqrt( (X - Xa)*(X - Xa)  + ( Y - Ya)*(Y - Ya) );

                // return the function Stress value
                return(Stress);
        }

}
```

9.5 Van Genuchten Functions (SEEP/W, VADOSE/W, SIGMA/W)

This section of code has several function in it that relate to the Van Genuchten conductivity and water storage functions. In addition, there is a function that varies the Ksat randomly across the soil profile.

Unlike the examples above, this code has a section with some general functions that do not have Calculate() methods in them. They are general in that they contain code that is needed more than once in this section so instead of repeating the code twice or more, the common code was made into its own public method that is then called within the other methods.

```
// There are two general functions that get used within other functions.
// These are not called directly by the solvers.  They are used in the
// other functions below.

Using system;

public class My_General_Functions
```

```
{
        // This is a C# random number generation function
        // It is used in other functions within this file.

        public static Random autoRand = new Random();

        //This first function takes a pressure and returns the Van_G K value
        // given the hard wired a,n,m and Ksat values.
        // It is a general function here because the same code is used
        // twice below and it is not necessary repeating it both times
        // if it is made public to all other function in this file.
        // Notice that it does NOT have a Calculate() method in it.  It will not
        // show up in Define View.  It will just be available to other function in this file.

        public static double Van_G_K_Unsat( double pressure, double fa, double fn, double fm,
                                double fKsat )
        {

                // returned K value
                double fKx;

                // temporary variables
                double fTemp1, fTemp2, fTemp3, fTemp4, fTemp5, fTemp6;

                if(pressure < 0.0) // if in the unsaturated side of the function
                {
                        double fSuction = Math.Abs (pressure);
                        fTemp1 = fSuction*fa;
                        fTemp2 = (Math.Pow((1.0 + Math.Pow(fTemp1, fn)), (fm/2)));
                        fTemp3 = Math.Pow(fTemp1, (fn-1));
                        fTemp4 = (1.0 + Math.Pow(fTemp1, fn));
                        fTemp5 = Math.Pow(fTemp4, -fm);
                        fTemp6 = Math.Pow((1.0 - fTemp3 * fTemp5), 2.0);
                        fKx = fKsat * (fTemp6/fTemp2);
                }
                else // use the user input Ksat if pwp are zero or positive
                        fKx = fKsat;

                return fKx;
        }

        // This is the second general function in this file.  It is called by other functions
        // below that do have a Calculate() method in them.

        public static double Van_G_VWC( double pressure, double fa, double fn, double fm,
                        double fPorosity, double fResidualWC )
        {
                double fWC, suction;      // returned K value
                double fTemp1, fTemp2; // temporary variables

                if(pressure < 0.0) // if in the unsaturated side of the function
                {
                        suction = Math.Abs (pressure);
                        fTemp1 = suction*fa;
                        fTemp2 = Math.Pow(  1.0 /  (1.0 + Math.Pow(fTemp1, fn)  ) , fm );
                        fWC = fResidualWC + (fPorosity-fResidualWC) * fTemp2;
```

```
                }
                else // use the user input porosity if pwp are zero or positive
                        fWC = fPorosity;

                return fWC;
        }
}  // end of general functions in this file

//*********************************************************************************************
// This function will let the Ksat of a conductivity function
// vary randomly one order of magnitude across the profile.
// It DOES have a Calculate() method so will be visible
// as an Add-In function in Define View.

public class Random_Van_G_K_Unsat : Gsi.Function
{
        public double KSat;
        public double alpha_one_over_pressure;  // Van G "a" parameter in units of 1/pressure
        public double n;                        // Van G "n" parameter, unitless
        public double m;                        // Van G "m" parameter, unitless
        double KRandomOffset;

        // this is called a constructor.  It will set values the first
        // time each instance of this function is called.
        // Here, it assigns a random number that will remain the same
        // throughout the analysis.

        public Random_Van_G_K_Unsat()
        {
                // returns a double between 0.0 and 1.0
                KRandomOffset = My_General_Functions.autoRand.NextDouble();
        }

        public double Calculate( double pressure )
        {

                double fKx = My_General_Functions.Van_G_K_Unsat(pressure,
                                alpha_one_over_pressure,n,m,Ksat);

                fKx = Math.Log10(fKx) ;

                // add up to 1 order of magnitude to K (which is Log at this point)
                if(KRandomOffset > 0.5)
                        fKx = fKx + KRandomOffset;
                else
                        fKx = fKx - KRandomOffset; // make 1 order of magnitude less that K

                return Math.Pow(10.0,fKx);
        }

}

//*********************************************************************************************

// sample function to return the unsat K function based on a Ksats as input by the user
public class Van_Genuchten_K_Unsat : Gsi.Function
```

```
{
        public double KSat;
        public double alpha_one_over_pressure; // Van G "a" parameter in units of 1/pressure
        public double n;                       // Van G "n" parameter, unitless
        public double m;                       // Van G "m" parameter, unitless

        public double Calculate( double pressure )
        {

                double fKx = My_General_Functions.Van_G_K_Unsat(pressure,
                                alpha_one_over_pressure,n,m,KSat);

                return fKx;
        }
}

//*********************************************************************************************

public class Van_Genuchten_VWC : Gsi.Function
{
        public double Porosity;
        public double alpha_one_over_pressure;  // Van G "a" parameter in units of 1/pressure
        public double n;                       // Van G "n" parameter, unitless
        public double m;                       // Van G "m" parameter, unitless
        public double Residual_WC;

        // constructor to initialize data so graph does not give an error in Define View
        public Van_Genuchten_VWC()
        {
                Porosity = 0.5;
                alpha_one_over_pressure = 0.1 ; // Van G "a" parameter in units of 1/pressure
                n = 1.5 ;                       // Van G "n" parameter, unitless
                m = 0.5 ;                       // Van G "m" parameter, unitless
                Residual_WC = 0.05 ;
        }

        public double Calculate( double pressure )
        {
                double fWC = My_General_Functions.Van_G_VWC(pressure,
                                alpha_one_over_pressure, n, m, Porosity, Residual_WC);
                return fWC;
        }
}
```

9.6 AIR/W Pa_Fix_as_Pinitial

The main type of boundary condition in AIR/W is air pressure. There are cases where air pressure is not constant on non-horizontal face in which case, the air pressure boundary condition depends on elevation and perhaps density associated with temperature changes. This Add-In will read in the air temperatures from the initial condition file and fix them as constant values throughout the transient analysis. This way, if the correct non-uniform air pressures are solved for in an initial condition file, they can be used as a BC subsequently.

```
using System;

// this function should be applied to a Pa vs time BC function and it will return the initial condition
```

```
// air pressure as the fixed Pa value for all time steps.  This will let you get the right Pa on a
// non horizontal surface.  It will be the equivalent of a constant air head.


public class Pa_Fix_as_Pinitial : Gsi.Function
{
	// just give a dummy message so the UI shows how function works.
	public double There_are_no_input_parameters;

	// declare the variable to return to the solver.  Its is not public so won't show up as a UI
	 // item.  An instance of it will be owed by each node the BC function is applied to.  It's
	// value will be retained until the function closes.

	double Pa;

	// constructor to initialize data.  This only runs when a function is first initialized.
	public Pa_Fix_as_Pinitial()
	{
		Pa = 0.0; // set Pa default to be atmospheric pressure
	}

	// x is the passed in time of the Pa vs Time function.  It is not used in this version as Pa is
	// returned constant for all time.
	public double Calculate( double x )
	{
		int ItNum;
		int nStepNum;

		// this will get live data from the solver.
		try
		{
			ItNum = (int)GetParam(Gsi.DataParamType.eStepIterationCount);
			nStepNum = (int)GetParam(Gsi.DataParamType.eStepNum);
		}
		catch
		{
			ItNum = 0;
			nStepNum = 0;
		}

		// first time called so set up Pa based on the read in initial condition file Pa for
		// this node.

		if(ItNum == 1 && nStepNum == 1)
		{
			// this will get the air pressure at this node.  A copy of this function
			// exists for each node it is applied as a BC to.
			Pa = GetParam(Gsi.DataParamType.eAirPressure);
		}

		// return the function Pa value
		return(Pa);
	}

}
```

9.7 SIGMA/W Von_Mises Constitutive Model

This is a complex example of a elastic-plastic based constitutive model Add-In in SIGMA/W. There are several methods in the following code that have been created because their code is used in multiple places. When code is repeated, it should be made a common method and called by its name and a set of parameters.

```
using System;


// This is a Von Mises yield criteria elastic plastic model.  It HAS the CalculateMatrix and
// UpdateStresses methods which MUST be present in an Add-In constitutive model.  .

public class Von_Mises_Model : Gsi.Function
{
        // declare global constants, these will appear in the Key In Materials Dialogue
        //for data entry
        public double Cu;
        public double E;
        public double Poisson;

        // These are used locally in this code and don't require input by the user
        double fLastYield;
        double PWP;

        // There are four local methods called inside this Von Mises model.  They are used
        // as part of the main two methods that required in the Add-In model.  Their usage
        // in the code below is highlighted in blue.

        // The first returns a 4x4 matrix for a gauss point with elastic properties only.
        // It takes a modulus and poissons ratio as input.

        public void CalcElasticCee(double youngsModulus, double Poisson,
                                        Gsi.Matrix mCee ) // returns mCee matrix
        {
                double fCOM  = youngsModulus / ((1.0 + Poisson)*(1.0 - 2.0*Poisson));
                double fCOM1 = 1.0 - Poisson;

                mCee[0,0] = fCOM*fCOM1;
                mCee[0,1] = fCOM*Poisson;
                mCee[0,2] = fCOM*Poisson;
                mCee[0,3] = 0.0;

                mCee[1,0] = fCOM*Poisson;
                mCee[1,1] = fCOM*fCOM1;
                mCee[1,2] = fCOM*Poisson;
                mCee[1,3] = 0.0;

                mCee[2,0] = fCOM*Poisson;
                mCee[2,1] = fCOM*Poisson;
                mCee[2,2] = fCOM*fCOM1;
                mCee[2,3] = 0.0;
```

```
        mCee[3,0] = 0.0;
        mCee[3,1] = 0.0;
        mCee[3,2] = 0.0;
        mCee[3,3] = 0.5*youngsModulus/(1.0+Poisson);
}

// This returns the stress invariants J2, J3 and Lode Angle and deviatoric stresses.  It
// takes as input x,y,z stress and shear stress..

public void CalcStressInvariants(Gsi.Matrix mStresses,
                        out Gsi.Matrix mDevStress, out double  fJ2, out double  fJ3,
                        out double  fTheta, out double fSigMean )
{
        double fRootJ2;
        double dSin3T;

        double fSigX = mStresses[0,0];
        double fSigY = mStresses[0,1];
        double fSigZ = mStresses[0,2];
        double fTauXY = mStresses[0,3];

        fSigMean = (fSigX+fSigY+fSigZ)/3.0;

        // deviatoric stresses
        mDevStress = mStresses - fSigMean;
        mDevStress[0,3] = mStresses[0,3];  // reset shear stress

        fJ2 = 0.16667 * (  Math.Pow(fSigX-fSigY,2.0) + Math.Pow(fSigY-fSigZ,2.0) +
                Math.Pow(fSigZ-fSigX,2.0)  ) + Math.Pow(mDevStress[0,3],2.0);

        fRootJ2 = Math.Pow(fJ2,0.5);

        double sx,sy,sz;

        sx = (2*fSigX - fSigY- fSigZ)/3.0;
        sy = (2*fSigY - fSigZ- fSigX)/3.0;
        sz = (2*fSigZ - fSigX- fSigY)/3.0;

        fJ3 = sx*sy*sz - sx*fTauXY*fTauXY;

        if(fRootJ2 == 0.0)
                dSin3T = 0.0;
        else
                dSin3T = -3.0*Math.Pow(3.0,0.5)*fJ3/(2.0*fJ2*fRootJ2);

        if(dSin3T >  1.0)
                dSin3T =  1.0;
        if(dSin3T < -1.0)
                dSin3T = -1.0;

        fTheta = Math.Asin(dSin3T) / 3.0;
}


// This returns the yield amount stress based on Von Mises.  It uses the
```

```
// stress invariant method coded above

public void Von_mises_Yield(Gsi.Matrix mStresses, out double fYield)
{
        Gsi.Matrix mDevStress = Gsi.Matrix.Zero(1,4);    // Temp Vector
        double fJ2, fJ3, fTheta, fSigMean;

        CalcStressInvariants(mStresses, out mDevStress, out fJ2, out fJ3, out fTheta,
                out fSigMean);

        double fq = Math.Pow(3*fJ2,0.5);

        // Check yield in the post iteration check stress algorithm
        fYield = fq-Math.Pow(3,0.5)*Cu;
}

// This returns the Von_Mises plastic flow vector.  This also uses the stress
// invariant method code above.

public void Von_mises_PlasticFlow(  Gsi.Matrix mStresses, out Gsi.Matrix mPlasticFlow)
{
        double fJ2, fJ3, fTheta, fSigMean;
        Double fTemp;

        Gsi.Matrix mDevStress = Gsi.Matrix.Zero(1,4);    // Temp Vector

        CalcStressInvariants(mStresses,out mDevStress,out fJ2,
                out fJ3, out fTheta, out fSigMean);

        fTemp = 0.5 / Math.Pow(fJ2,0.5);
        mDevStress *= fTemp;
        mDevStress[0,3] *= 2.0;

        fTemp = Math.Pow(3.0,0.5);

        mPlasticFlow = mDevStress*fTemp;
}


// These are the two REQUIRED methods that MUST exist in a user constitutive model.
// The first returns the [C] matrix that will get put into the EPM at each iteration prior to
// SOLVING the equations

// The second computes the new stress state after the incremental displacements and
// strains are computed in SOLVE at each iteration.

public void CalculateMatrix( Gsi.Matrix mCee )
{

        CalcElasticCee(E, Poisson, mCee); // compute elastic [C]

        int ItNum = (int)GetParam(Gsi.DataParamType.eStepIterationCount);

        PWP = GetParam(Gsi.DataParamType.ePWP);
        double Sx = GetParam(Gsi.DataParamType.eXTotalStress) - PWP;
        double Sy = GetParam(Gsi.DataParamType.eYTotalStress) - PWP;
```

```
            double Sz = GetParam(Gsi.DataParamType.eZTotalStress) - PWP;
            double Sxy = GetParam(Gsi.DataParamType.eXYShearStress);

            Gsi.Matrix mCurrentStress = Gsi.Matrix.Zero(1,4);
            // current solution effective stress state prior to adding the current
            // incremental stress

            mCurrentStress[0,0] = Sx;
            mCurrentStress[0,1] = Sy;
            mCurrentStress[0,2] = Sz;
            mCurrentStress[0,3] = Sxy;

            if(ItNum > 1)
            {
                  if(fLastYield > 0.0)
                  {
                        Gsi.Matrix mdFdSig = Gsi.Matrix.Zero(1,4);
                        Gsi.Matrix mdGdSig = Gsi.Matrix.Zero(4,1);

                        Gsi.Matrix mTempVRow = Gsi.Matrix.Zero(1,4);
                        Gsi.Matrix mTempVCol = Gsi.Matrix.Zero(4,1);
                        Gsi.Matrix mCep = Gsi.Matrix.Zero(4,4);

                        Von_mises_PlasticFlow(mCurrentStress, out mdFdSig);

                        mdGdSig = Gsi.Matrix.Transpose(mdFdSig);

                        mTempVCol = mCee * mdGdSig  ;

                        mTempVRow = Gsi.Matrix.Transpose(mTempVCol);

                        double fTemp = Gsi.Matrix.Dot(mdFdSig,mTempVCol);

                        if(Math.Abs(fTemp)  < 1e-10) fTemp = 1.0e-10;

                        fTemp = 1.0/fTemp;

                        mCep = Gsi.Matrix.TensorCross(mTempVCol, mTempVRow);

                        mCep *= fTemp;

                        //Form plastic constitutive matrix.
                        mCee = mCee - mCep;  // [Cee-Cep]
                  }
            }
}

public void UpdateStresses( Gsi.Matrix mIncStrain, ref Gsi.Stresses Stresses)
{
            Gsi.Matrix mIncStress = Gsi.Matrix.Zero(1,4);
            Gsi.Matrix mCee = Gsi.Matrix.Zero(4,4);
            Gsi.Matrix mStartStress = Gsi.Matrix.Zero(1,4);
            Gsi.Matrix mUpdatedStress = Gsi.Matrix.Zero(1,4);

            Gsi.Matrix mStressYield = Gsi.Matrix.Zero(1,4);
            Gsi.Matrix mExcessStress = Gsi.Matrix.Zero(1,4);
```

```
Gsi.Matrix mPlasticStress = Gsi.Matrix.Zero(4,1);
Gsi.Matrix mPlasticStressT = Gsi.Matrix.Zero(1,4);
Gsi.Matrix mPlasticStrain = Gsi.Matrix.Zero(1,4);
Gsi.Matrix mPlasticStrainT = Gsi.Matrix.Zero(4,1);
Gsi.Matrix mdFdSig = Gsi.Matrix.Zero(1,4);
Gsi.Matrix mdGdSig = Gsi.Matrix.Zero(4,1);
Gsi.Matrix mTempVec = Gsi.Matrix.Zero(1,4);

Double fYield = 0.0;
Double fYieldStart = 0.0;
Double fReductionFactor = 1.0;
Double fTemp = 0.0;
Double fDLambda;

int i, iSubInc;
iSubInc = 20;  // we will subdivide any yielded strains into smaller pieces
               // so that we can compute the excess stresses that need
               // to be subtracted from the elastic overshoot to get us
               // back to the yield surface.

int ItNum = (int)GetParam(Gsi.DataParamType.eStepIterationCount);

// form the elastic Cee matrix as it is needed for stress calculations
CalcElasticCee(E, Poisson, mCee);

// compute elastic Ce used to know the stress increment
// {IncStress} = [Cee]x{IncStrain}   This is the full incremental stress
// based on the elastic load

mIncStress = mIncStrain * mCee;

mStartStress[0,0] = GetParam(Gsi.DataParamType.eXTotalStress)   - PWP;
mStartStress[0,1] = GetParam(Gsi.DataParamType.eYTotalStress)   - PWP;
mStartStress[0,2] = GetParam(Gsi.DataParamType.eZTotalStress)   - PWP;
mStartStress[0,3] = GetParam(Gsi.DataParamType.eXYShearStress) ;

// compute updated stresses with full elastic increment applied.
// This will be reduced later if there has been yield
mUpdatedStress = mStartStress + mIncStress;

// check yield with starting stresses and then with updated stresses
Von_mises_Yield(mStartStress,out fYieldStart);
Von_mises_Yield(mUpdatedStress,out fYield);

// find reduction factor
if(fYield < 0.0)                        // still elastic
        fReductionFactor = 0.0;
else if(fYieldStart >= 0.0)              // was plastic so stays plastic
        fReductionFactor = 1.0;
else                                    // part elastic, part plastic
{
        if(fYield*fYieldStart > 0)
                if(fYield > 0)
                        fReductionFactor = 1.0;
                else
```

```
                        fReductionFactor = 0.0;
        else
                fReductionFactor = fYield/(fYield-fYieldStart);
}

mStressYield = mStartStress + mIncStress*(1.0-fReductionFactor);

if( fReductionFactor > 0.0)
// there are some plastic strain induced stresses that need to be reduced
{
        fLastYield = 1.0;

        mExcessStress = mIncStress*fReductionFactor/iSubInc;

        // find plastic strain for the given total strain increment, compute the
        // elastic stress that would have caused this strain,
        // then reduce the initially computed stress increment by this plastic
        //stress amount (there can be no change in stress for plastic strain so
        // take it off)
        for(i=0;i<iSubInc;i++)
        {
                Von_mises_PlasticFlow(mStressYield, out mdFdSig);
                        mdGdSig = Gsi.Matrix.Transpose(mdFdSig);

                fDLambda = Gsi.Matrix.Dot(mdFdSig, mExcessStress);

                mTempVec = mCee * mdGdSig ;

                fTemp = Gsi.Matrix.Dot(mdFdSig,mTempVec);

                fDLambda = fDLambda / fTemp;

                mPlasticStrain = mdFdSig * fDLambda;

                mPlasticStrainT = Gsi.Matrix.Transpose(mPlasticStrain);

                mPlasticStress = mCee * mPlasticStrainT;

                mPlasticStressT = Gsi.Matrix.Transpose(mPlasticStress);

                // reduce actual incremental stresses by plastic amount
                mStressYield += (mExcessStress-mPlasticStressT);

        }
        mUpdatedStress = mStressYield + PWP;
}
else
{
        mUpdatedStress = mStressYield + PWP;
        fLastYield = 0.0;
}

//  These are the five values that are returned by this UpdateStresses method.
//  The solver is expecting valid values for all of these.  These will be used
//  with the previous iteration stresses to determine the unbalanced load for
//  the next iteration.
```

```
            Stresses.x = mUpdatedStress[0,0];
            Stresses.y = mUpdatedStress[0,1];
            Stresses.z = mUpdatedStress[0,2];
            Stresses.xyShear = mUpdatedStress[0,3];
            Stresses.voidRatio = 0.0;   //   this is optional to return from this method.
      }
}
```

9.8 SLOPE/W Cohesion using Temperature

Here is a detailed example for a SLOPE/W where the strength of the ground depends on temperatures that are read in from a previously solved TEMP/W analysis. The SLOPE/W Add-In creates a local data table for nodal temperature data and then can access the temperature at any X,Y location when it needs to know strength at the base of any slice.

```
public class Cohesion_Freeze_Thaw : Gsi.Function
{
public double Cu_frozen;
public double Cu_thawed;
public String Freezing_On_Off;
public String Linked_Analysis_Name;
public double Test_temperature;

double Temperature;

static Gsi.DataTable tbl; // = new Gsi.DataTable();
static System.Object lockThis = new System.Object();
static int iLastStepLoaded = 0;

System.String sAnalysis_Name;

public double Calculate( double Passed_in_X )
{
      // read in Y-coordinate at slice base center
      double BaseCenterY;
      double BaseCenterX;
      int nStep;

      try
      {
            // this will get live data from the solver.
            BaseCenterY = GetParam(Gsi.DataParamType.eYCoord);
            BaseCenterX = GetParam(Gsi.DataParamType.eXCoord);
      }
      catch // exception thrown - we must not be in the solver!
      {
            // called in the function graph window of GeoStudio.
            BaseCenterY = 0;
            BaseCenterX = Passed_in_X;
      }

      sAnalysis_Name = Linked_Analysis_Name ; // "3 Transient Thermal";

      if(((Freezing_On_Off == "On") ||(Freezing_On_Off == "on")))
      {
```

```
            // load table on first slice only
            try
            {
                  nStep = (int)GetParam(Gsi.DataParamType.eStepNum);
            }
            catch
            {
                  nStep = iLastStepLoaded;
            }

            try
            {
                  // Be careful to lock here! Add-Ins may be run
                  // in multiple threads, so protect shared data
                  lock(lockThis)
                  {
                        if ( iLastStepLoaded != nStep )
                        {
                              tbl =
                                    Gsi.DataTable.Load(
                                          sAnalysis_Name,
                                          Gsi.TableType.eNodeTable,
                                          nStep );
                              iLastStepLoaded = nStep;
                        }
                  }
            }
            catch
            {
            }

            if(nStep != iLastStepLoaded) // so solver is calling this
                  Temperature =
                        Gsi.Mesh.ComputeNodalValueAtXY(
                              tbl,
                              BaseCenterX,
                              BaseCenterY ,
                              Gsi.DataParamType.eTemperature
                              );
            else
                  Temperature = Test_temperature;
      }

      if(
            ((Freezing_On_Off == "On") ||(Freezing_On_Off == "on")) &&
             (Temperature < 0)
            )  // use frozen Cu for frozen ground
      {
            return (Cu_frozen);
      }
      else
            return(Cu_thawed);
}

}
```