The University of Texas at Austin

Cockrell School of Engineering

Civil, Architectural, and Environmental Engineering

Introduction to Object-Oriented Programming and Related Applications in Commercial Structural Engineering Software

Departmental Report

Alex Dundore

Table of Contents

General Guide to Programming in Commercial Structural Analysis Software

1 - Introduction to Object-Oriented Programming

Most engineers studying structural engineering have some level of experience in programming tools to aid in numerical analysis. The most common of these tools in the academic world is MATLAB. Programs like MATLAB are incredibly helpful for the purpose they were designed for, which is doing a large number of numerical calculations as efficiently as possible. These types of tools utilize a programming syntax that is relatively easy for engineers to learn and use without having to devote a substantial amount of time to master it. Numerical programming is certainly a powerful skill to have in your back pocket as an engineer, but it lacks functionality in certain areas that more traditional programming languages can take full advantage of.

Although the use of tools like MATLAB is often considered "programming", it is not quite the same as programming in a classical sense. Common programming languages like Python or C are different in the aspect that they are object-oriented. Object-oriented programming is a classification based on the concept of "objects" in code. Objects are items within the code that can contain both data and additional code. The data is stored in the form of "fields" (often referred to as "properties"). These simply contain information about a specific object. For example, properties of an array object might be its length or its maximum value. The additional code contained within the objects is in the form of "procedures" (often referred to as "methods"). These methods are specific to a certain type of object and have limitless uses. For example, a method of an array object might be to append a new value to the array or to return the index of the maximum value. The examples of properties and methods highlighted here are rather elementary, but should give a general idea of how they work.

Numerical tools like MATLAB also have a limited ability to communicate with outside programs, whereas typical programming languages like Python or C do not have these limitations. In most programming languages, it is fairly simple to interact with other programs running on a machine, control parts of the operating system, or even connect to the internet.

There are many other differences between numerical analysis tools such as MATLAB and classical object-oriented programming languages, but that is outside the scope of this guide. For purposes of programming in the structural engineering industry, leveraging the abilities to use objects to store/access data and to interface with other programs running on a machine can go a long way.

The remainder of this section will provide a general overview of object-oriented programming practices in Visual Basic for Applications and examples of its practical use with commercial structural engineering software such as SAP2000. Descriptions and examples will assume that the reader has some basic experience in numerical programs like MATLAB and will thus relate ideas in object-oriented programming to analogous practices in MATLAB. This guide is certainly not an exhaustive lesson on all useful practices in VBA, but is meant to introduce the reader to the concepts and give them enough intuition that they should be able continue learning more on their own.

This guide was created assuming that the user is on a machine running Microsoft Windows. If a different operating system is used, the details might differ slightly from those presented here.

2 - Using Visual Basic for Applications (VBA) within Microsoft Excel

Visual Basic for Applications (often abbreviated as VBA) is a programming tool created by Microsoft that can run user-created code in an object-oriented form. VBA is embedded within a host program and comes built-in with many programs that Microsoft creates, including Excel [1]. Using VBA within Excel will be the main focus of this guide.

To use VBA in Excel, it must first be enabled. To do this, go to File > Options within Excel. On the "Customize Ribbons" section, ensure that the "Developer" tab is enabled, then press OK. The new Developer tab will appear in the Excel ribbon. Within the ribbon, clicking on the "Visual Basic" button will open VBA in a new window.

It should be noted that in order to save VBA scripts with the Excel document, the workbook will need to be saved as an "Excel Macro-Enabled Workbook" file type (.xlsm) rather than the typical "Excel Workbook" file type (.xlsx).

2.1 - General Layout of VBA

The default Visual Basic window within Excel is shown in Figure 1. On the left, the Project Browser is shown. This window shows all of the open Excel documents as well as their corresponding worksheets. This is also where the user-created VBA scripts (also called "Macros") will appear.

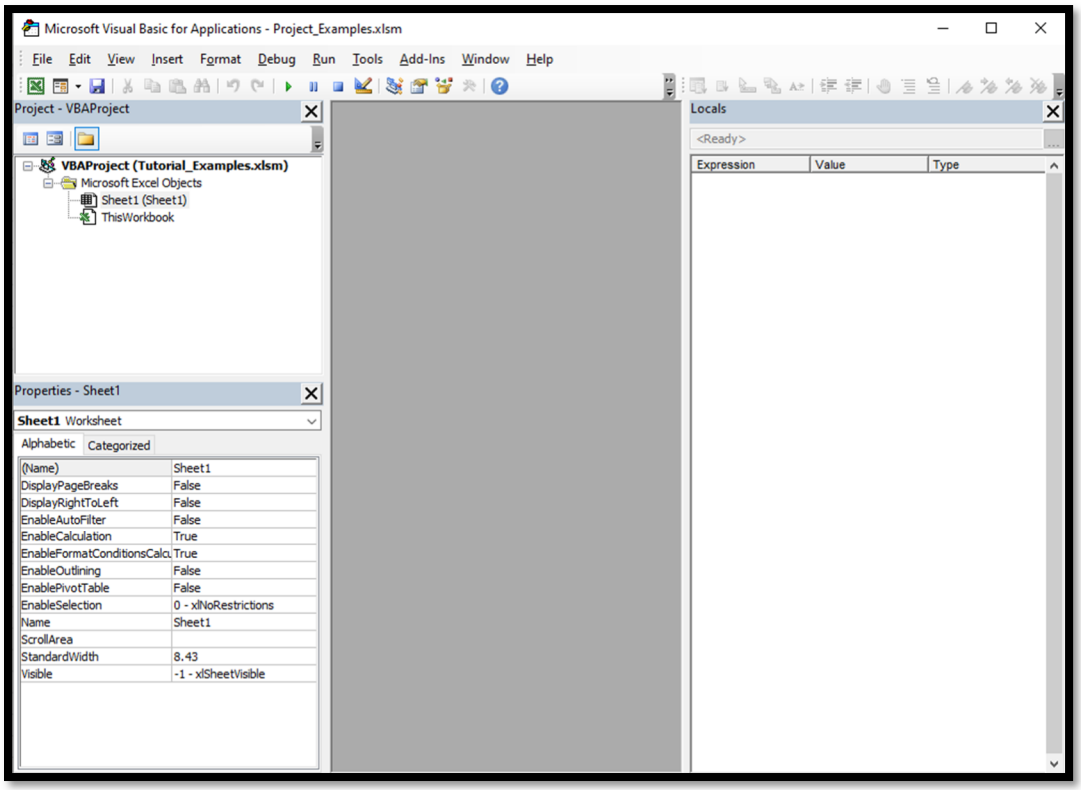


Figure 1: Default Visual Basic for Applications Window

Below the Project Browser is the Properties Browser. This shows the properties of the current selection in the Project Browser. This is not very important for the purposes of this guide.

On the right is the Locals Window. If this does not appear by default, it can be shown by selecting View > Locals Window. This will track all of the variables and their values that exist in the current script during debugging. It is very similar to the workspace browser in MATLAB.

The empty area in the middle will be used to edit macros, but will be blank when it is first opened.

2.2 - Modules and the Distinction between Local and Global Variables

Macros, or scripts, in VBA are created within Modules. In order to create a module in your project, right click on an empty area in the Project Browser and select Insert > Module. A new module will be created in your project and appear in the Project Browser as shown in Figure 2.

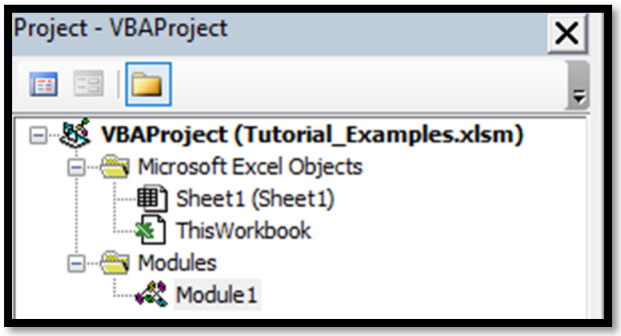


Figure 2: Creation of New Module

The name of the new module can be edited in the Properties window (it will be called “Module1” by default). Additionally, you will see that the area in the center that was once blank has been populated with an area to edit code. This is where the code for your macros can be created.

An important distinction should be made that relates variables to certain modules. In MATLAB, the important variables will exist within the single script you are running, which is usually the only script of interest. In VBA, code is usually broken into separate modules. Because of this, two different variable types exist: Local and Global variables. Local variables are those that only exist within the macro (and corresponding module) that is currently running. When a variable is created within a macro, it will be a local variable by default. These local variables will be the ones shown in the Locals window in the development environment. Once the execution of this macro has finished, all of the local variables will be wiped from the system’s memory. Global variables, on the other hand, are not wiped from memory and can be used between modules once the variable has been created. Global variables must be declared in a particular way in order to distinguish them from Local variables.

2.3 - Subroutines and Variable Types

When creating a macro within a module, you will have to define a subroutine that contains the code to be run. Subroutines are pieces of code that perform specific tasks in the code and do not return values. In order to create one, a certain syntax must be used in the code editing window. A subroutine is indicated by using the word “Sub” followed by the name of your subroutine with closed parenthesis at

the end. The end of the subroutine is marked by the words "End Sub". Any code between the declaration and ending of the subroutine will be run when the macro is executed. An example of this syntax is shown in Code Block 1. Note that the body of the code is represented by a comment, shown in green.

```
Sub MySubroutine()

    ' Enter Code Here

End Sub
```

Code Block 1: Defining a Subroutine

Once a subroutine is created, the macro can be run by pressing the "Run Macro" button (shaped like a play button) or by pressing F5. The execution of the body of the macro will work just like MATLAB, going line by line until the end of the subroutine is reached.

The main body of the macro will inevitably contain many variables, as in programs like MATLAB. Working with variables in object-oriented programming languages is quite different than in MATLAB, though. In VBA (and other object-oriented programming languages), proper syntax dictates that you must declare your variables before you assign a value to them. Although this is not required in all situations, it will significantly help how your code runs and give you more control over what data types your variables use.

To declare variables, one line of code is required for declaration (or "dimensioning") and another separate line will be used for assigning the newly created variable a value. The typical declaration syntax includes the "Dim" statement, the variable name, and the data type. The variable can then be assigned a value pertaining to the specific data type in a new line. An example is shown in Code Block 2.

```
Dim x As Integer
x = 5
```

Code Block 2: Typical Variable Declaration Syntax

Common data types in VBA that would be useful for engineering applications are Boolean, Integer, Double (floating point numbers), and String. The type "Variant" can be used when declaring a variable that can be used for both numerical and non-numerical values. There are certain situations where this can be advantageous, which will be discussed in more detail later.

Arrays of certain variable types can also be created in VBA, but their size must be known during declaration of the variable. An array is indicated during declaration by including closed parenthesis after the variable name, where the size is indicated within the parenthesis. In VBA (and all other classical programming languages), the first index is usually considered to be zero, rather than MATLAB's convention of one. Also, the values of each index are typically set one at a time.

If the array size is not initially known, a blank array can be created and then the "ReDim Preserve" statement can be used to re-dimension the size of the array later and preserve the current values; just be aware that this can use substantial amounts of memory with large arrays. Object-oriented languages

are not usually optimized for numerical calculations the way that MATLAB is. Examples of two types of array creation discussed are shown in Code Block 3.

```
' Case where initial size is known
Dim MyFloats(0 To 2) As Double
MyFloats(0) = 0.354
MyFloats(1) = -2.5
MyFloats(2) = 5068.9

' Case where initial size not known, so initially declare empty array
Dim MyUnknownFloats() As Double
n = 2
' Redimension the array to new size
ReDim Preserve MyUnknownFloats(0 To n)
MyUnknownFloats(0) = 0.354
MyUnknownFloats(1) = -2.5
MyUnknownFloats(2) = 5068.9

' Can also redimension array to change size again
ReDim Preserve MyUnknownFloats(0 To 3)
MyUnknownFloats(3) = -0.3
```

Code Block 3: Defining Arrays

2.4 - Conditional Operators & Loops

Conditional operators and loops in VBA work very similarly to both MATLAB and other classical programming languages. The only notable difference is syntax. Examples of a conditional statement, a for-loop, and a while-loop are demonstrated in Code Block 4. Further information, including specifics on conditional operator sign conventions, is readily available online.

```
' Conditional Statement Example
If x >= 1 And y = 0 Then
    ' Do something
End If

' For Loop Example
For i = 0 To 5
    ' Do something
Next i

' While Loop Example
Dim j As Integer
j = 0
Do While j <= 10
    ' Do something
    j = j + 1
Loop
```

Code Block 4: Conditional Operators and Loops

2.5 - Creating Functions

Functions in VBA work in a similar way to MATLAB. They should be created in a separate module than where the main script is executed. Functions are indicated and ended by using the syntax "Function" and "End Function", similarly to subroutines. When defining a function, the function name will be defined and followed by parenthesis that contain the input variables and their respective data types. The function name should be followed by the data type of the output; it is typical to define the function output as variant, then assign a data type within the function itself. An example of a function and its use in a separate module are shown in Code Block 5. Given an array and a number as input, it will manually increment all of the values in the input array by that number. The output from the Locals window after the subroutine has run is shown for reference.

```
Function IncrementArray(InputArray() As Double, Increment As Double) As Variant

' Find length of input array using "UBound()"
Dim length As Integer
length = UBound(InputArray)

' Dimension a new array of the same length
Dim output() As Double
ReDim Preserve output(0 To length)

' Add increment to each value in the original array
For i = 0 To length
    output(i) = InputArray(i) + Increment
Next i

' Assigning the output to the function name
IncrementArray = output

End Function

Sub MySubroutine()

' Creating initial "MyNumbers" array
Dim MyNumbers(0 To 2) As Double
MyNumbers(0) = 1.02
MyNumbers(1) = 6.68
MyNumbers(2) = 104.01

' Using function and assigning
' output the variable "MyNewNumbers"
MyNewNumbers = IncrementArray(MyNumbers, 2.5)
```

Locals

VBAProject.Module1.MySubroutine

Expression	Value
⊞ Module1	
⊟ MyNumbers	
MyNumbers(0)	1.02
MyNumbers(1)	6.68
MyNumbers(2)	104.01
⊟ MyNewNumbers	
MyNewNumbers(0)	3.52
MyNewNumbers(1)	9.18
MyNewNumbers(2)	106.51

Code Block 5: Implementation of a Simple Function in VBA

It is important to note that the user-defined function and the subroutine that uses it should be compartmentalized into separate modules for organizational purposes, similar to in MATLAB.

2.6 - Introduction to Objects and Interacting with Spreadsheets

In addition to performing numerical computations within a macro, it is possible to use VBA to interact with a spreadsheet in the host file that it is a part of. Some functionalities include gathering data from spreadsheets, writing data to spreadsheets, and performing operations within the spreadsheet. Custom functions for use by cells in the worksheet can also be created, but that is outside the scope of this guide.

Up to this point, the demonstrations of the capabilities of VBA have not deviated much from that of programs like MATLAB. Before describing how to interact with spreadsheets, however, it is necessary to discuss the concept of objects in more depth. In object-oriented programming, an object can be generally described as a data type that has extra functionality, which can come in the forms of properties and methods. Properties contain information about certain attributes of the object (length might be a property of an array, for example). Methods allow certain actions to be taken that relate to the object (a method to add an additional index to an array, for example). Note that arrays are not actually object data types in VBA; they are being used as an example simply because they are a familiar concept to the reader. Also, note that it is possible to create your own custom object types that contain corresponding properties and methods; these are called classes. Custom classes will not be covered in this guide, but they can be useful in certain situations.

When discussing objects, it is also important to discuss the concept of an instance and how it differs from an object type. If you are new to object-oriented programming but have experience with tools like MATLAB, this will likely be somewhat intuitive. An object type is the data type, which contains a set of properties and methods. An instance of that object type is a specific existing variable that those properties and methods can be used on. Using an analogy to typical variables might make it easier to understand. For example, the data type “Double” could be compared to the object type, and “x = 1.25”, “y = 5.869”, and “z = 8.1” could be compared to instances of that object type.

The ways to access properties and methods of an object are more or less the same across all object-oriented programming languages, VBA included. Properties of an object instance are accessed by typing the object instance and adding a “.” followed by the name of the property of interest. For example, if “x” was an instance of the aforementioned hypothetical array object, “x.Length” might return the number of elements in the array. Methods can be accessed in a similar way, but you must also add parenthesis following the method name as well as any arguments that the method requires. For example, “x.Append(2.43)” might add the element “2.43” to the array. Required arguments will vary for each method, and not all methods require arguments. For example, “x.DeleteLast()” might remove the last element of the array “x”, but would not require an argument (however, the parenthesis must still be used). This might sound abstract, but it will become more apparent after seeing the following example of real objects within VBA.

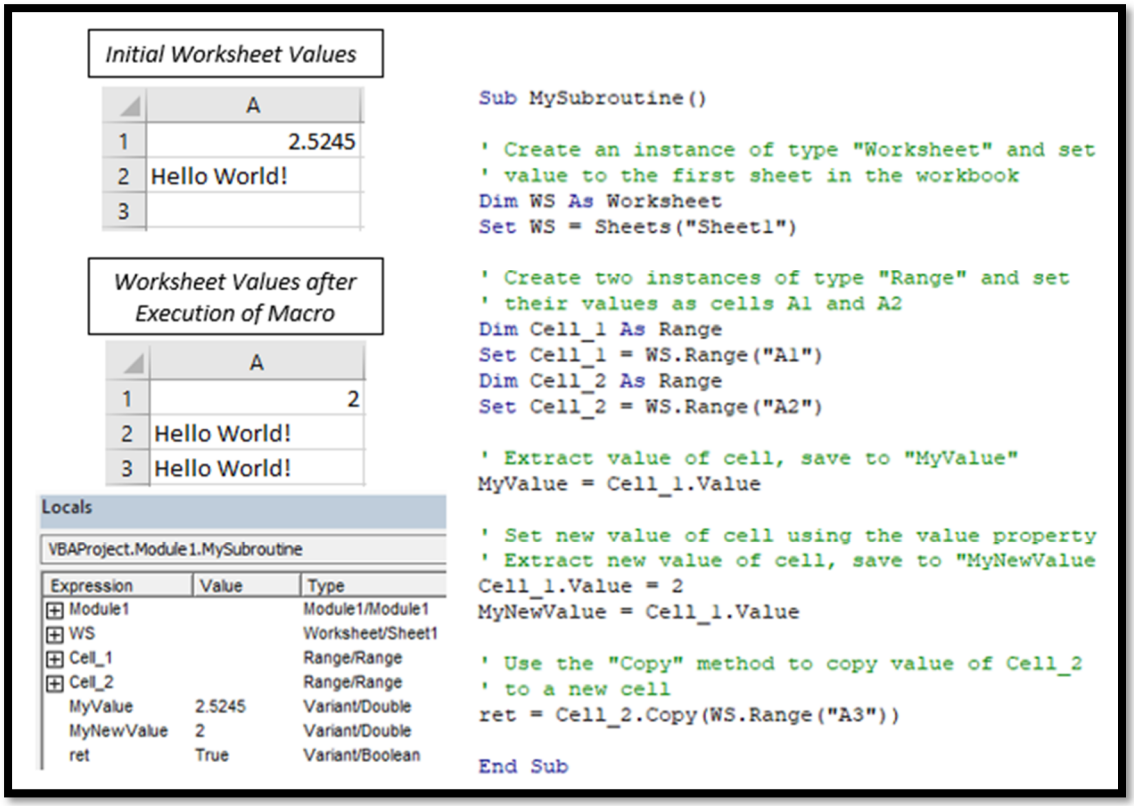
In order to associate a worksheet with a macro in VBA, you can first create an instance of the sheet of interest in the macro. The first section of code in Code Block 6 does this. The first line declares “WS” as a Worksheet object. The second line assigns the worksheet “Sheet1” to the instance “WS”. The use of the “Set” statement is something specific to Excel, and the details are not important other than the fact that it must be used in this fashion with sheets.

The second section of code declares “Cell_1” and “Cell_2” as Range objects. Range objects can be used to refer to specific cells or ranges of cells within a worksheet (for example, instead of “A1”, “A1:C5”

could be entered as an argument to select a range of multiple cells). The newly created variables are then assigned specific cells in the worksheet "WS". The reason that "WS.Range()" must be used to designate the cell is because Ranges are also properties of worksheets. Without referring to the worksheet of interest first, the code would not know which worksheet to gather that Range from.

At this point, "Cell_1" and "Cell_2" are variables within the macro that are pointing to actual cells in the Excel worksheet. Now, it is simple to take advantage of the properties and methods that belong to the Range object type. In the example, the value of Cell_1 is extracted and copied to a new variable called "MyValue" by using the ".Value" property of "Cell_1", which is a Range object type. As made apparent by the Locals window, the value of the variable "MyValue" has taken on the value of cell A1 in the worksheet. The values of properties can also be edited within the macro. For example, the value of cell A1 in the worksheet is changed by setting the "Cell_1.Value" equal to a new value. It can be seen that this change is reflected in the worksheet after the code was executed.

The last block of code demonstrates the use of the ".Copy()" method of Range objects. This method will copy the value of a cell and assign it to the cell passed as the method's argument. In this case, the value of "Cell_2" was assigned to cell A3 using the ".Copy()" method. This is apparent by examining the worksheet after the code was executed. Note that the variable "ret" was set equal to the method; this is because most methods will return a value after they are executed (hence "ret" – short for "return"). In this case, the ".Copy()" method returns the Boolean "True" if the method is executed without error.

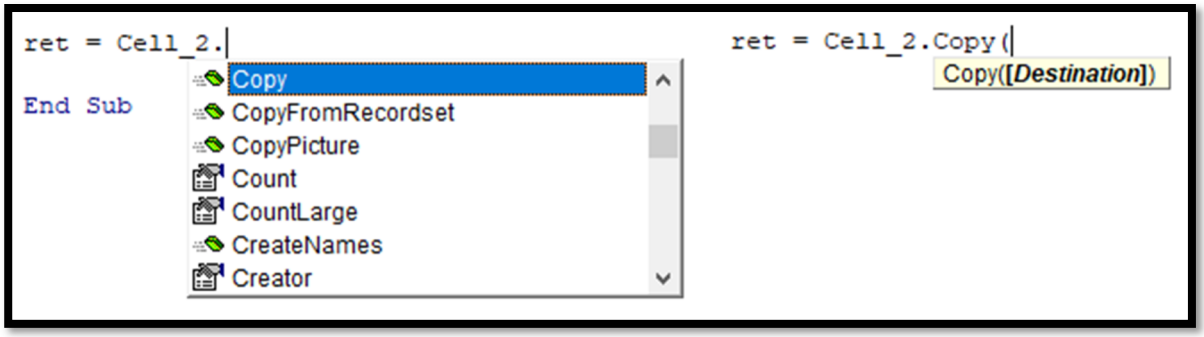


Code Block 6: Interacting with Excel Worksheets

2.7 - Using the Auto-Complete Suggestions in VBA

When programming in VBA, the development environment will automatically suggest methods and properties that can be used with the current object instance. In Code Block 7, the suggestions are shown for "Cell_2" of object type Range after the dot has been typed. This is a list of applicable methods (indicated by green icons) and properties (indicated by grey icons) for the instance. This can be very helpful when learning to program.

If you select a method, the development environment will also give you hints for what the arguments should be. In Code Block 7, after the ".Copy()" function has been chosen and the first parenthesis started, the program is indicating that the argument of the chosen method is a destination cell to copy the cell value to. If the selected method requires multiple arguments, there will be multiple arguments shown in the hint to reflect this. These features are common to most development environments for object-oriented programming. If you need more information than what is given by the development environment, it is recommended that you should examine the documentation for the method/property online.



Code Block 7: Auto-Complete Suggestions and Argument Hints

2.8 - Implementing Macros via Buttons in Worksheets

Once a macro has been created, it can be run using the "Run Macro" button within the VBA development environment. This is not convenient if a tool is being created for use specifically within the worksheet. To better suit this application, buttons can be created within the worksheets themselves that will execute certain macros when pressed.

In order to add a button, under the "Developer" tab on the Excel ribbon, select Insert > Form Controls > Button, then draw the button in the worksheet. After drawing the button, Excel will prompt the user to select a macro (a subroutine to be particular) that will execute when the button is pressed. By right clicking the button, you can edit the button text, resize the button, and move it around. Left clicking the button will run the assigned macro.

2.9 - Various Tips for Debugging and General Workflow Improvement

Visual Basic for Applications can be a difficult tool to master, especially for someone who does not have a background in object-oriented programming. There are some general tips that can help out during the code development process.

One fairly obvious tip is to regularly comment your code using the apostrophe character. This is helpful for any programming language if you are returning to code that was written long ago or when trying to decipher someone else's code.

When debugging a script to find where issues are occurring, it is helpful to add breakpoints to certain lines in your code. These stop the macro from executing past the indicated line until the user manually continues the execution by pressing the play button again. When the code has stopped running, it can help the user check the variables in the Locals window to see if everything is running as expected. Breakpoints can also help to pinpoint where errors are occurring in the code. It should also be noted that macros will clear local variables after they have been run; if you want to check the values of local variables after a script has run, you must put a breakpoint in at the "End Sub" line in the macro. To add a breakpoint, click in the grey area to the left of a line of code and a red dot will appear. To get rid of the breakpoint, click the red dot again.

The Watch window is another debugging tool that can be used in addition to the Locals window. It is different in the aspect that it will only watch specific variables that are manually specified by the user. It can also show the values of Global variables in addition to Local variables. Enable the Watch window by selecting View > Watch Window within the VBA development environment.

It is possible to execute multiple macros at one time using another macro. This is especially helpful if you'd like to keep code separated into multiple modules in order to stay organized, but retain the ability to run them in sequence as if they were part of one large code. This can be done by implementing the "Call" statement. A simple example of this is shown in Code Block 8.

```
Sub MyBigMacro()

Call MyFirstMacro
Call MySecondMacro
Call MyThirdMacro

End Sub
```

Code Block 8: Calling Multiple Macros from Another Macro

A final tip is to search online for help on object types and the use of their properties/methods if you are having trouble. Documentation for VBA [2] is very extensive on the internet and there are many forums online that are full of questions people have had about VBA. If you are having trouble with something, chances are someone else has already had the same problem and devised a solution on an online forum.

3 - Introduction to the SAP2000 Open Application Programming Interface

SAP2000 is a finite element analysis program for structural engineering and design created by Computers and Structures, Inc. (CSI) [3]. It is one of the most widely used commercial structural analysis programs in the industry. However, modeling large and repetitive structures in commercial software such as SAP2000 can be tedious and time-consuming. Using the Open Application Programming Interface (OAPI) that CSI has created for SAP2000 can significantly speed up these processes if the user has a basic knowledge of object-based programming.

3.1 - What is an Application Programming Interface?

An application programming interface (API) is an interface that allows two external programs to communicate with one another and lays rules for them to do so. Within the API, the software creators define objects, methods, and data types that correspond to the program that the API is a part of, which can be used and manipulated in an external programming language.

In the case of SAP2000's API, CSI has made it fairly simple for users to create scripts in programming languages like C# and VBA that will carry out commands within an open instance of SAP2000. Anything that can be done within the SAP2000 GUI by hand can be done automatically with code using the API. A macro in VBA containing thousands of commands for SAP2000 to carry out might take mere seconds to run, as opposed to hours for a user to manually carry out. For example, a code could be written to create the geometry of a 50 story building in VBA; when run, the geometry would appear in SAP2000 within seconds. Modeling the same building by hand would take much longer.

The benefits of utilizing a tool like this become immediately apparent when some sort of analysis with a parameterized structure needs to be done; this often takes the form of optimization problems. It is very simple to create code that can instantly create complex geometry in analysis software given simple parameters (ex: bay width, number of stories, story height, etc.). The results of an analysis could also be extracted from the software and post-processed within the code. Because all the modeling and post-processing is being done externally with code, iterating on the design becomes a simple task.

3.2 - Navigating the SAP2000 Open API Documentation

The documentation for the SAP2000 OAPI comes with the installation of the program in the form of a ".chm" (Compiled HTML Help) file. This can be found in the same program files directory as the SAP2000 application itself; the file is called "CSi OAPI Documentation.chm". This form of documentation is particularly helpful for new API users, as the document is organized into easily navigatable sections and is fully searchable. Figure 3 shows the documentation file with the OAPI functions section expanded. It may be noticed that the built-in functions of the API are organized in a similar fashion to the main toolbar in the SAP2000 graphical user interface.

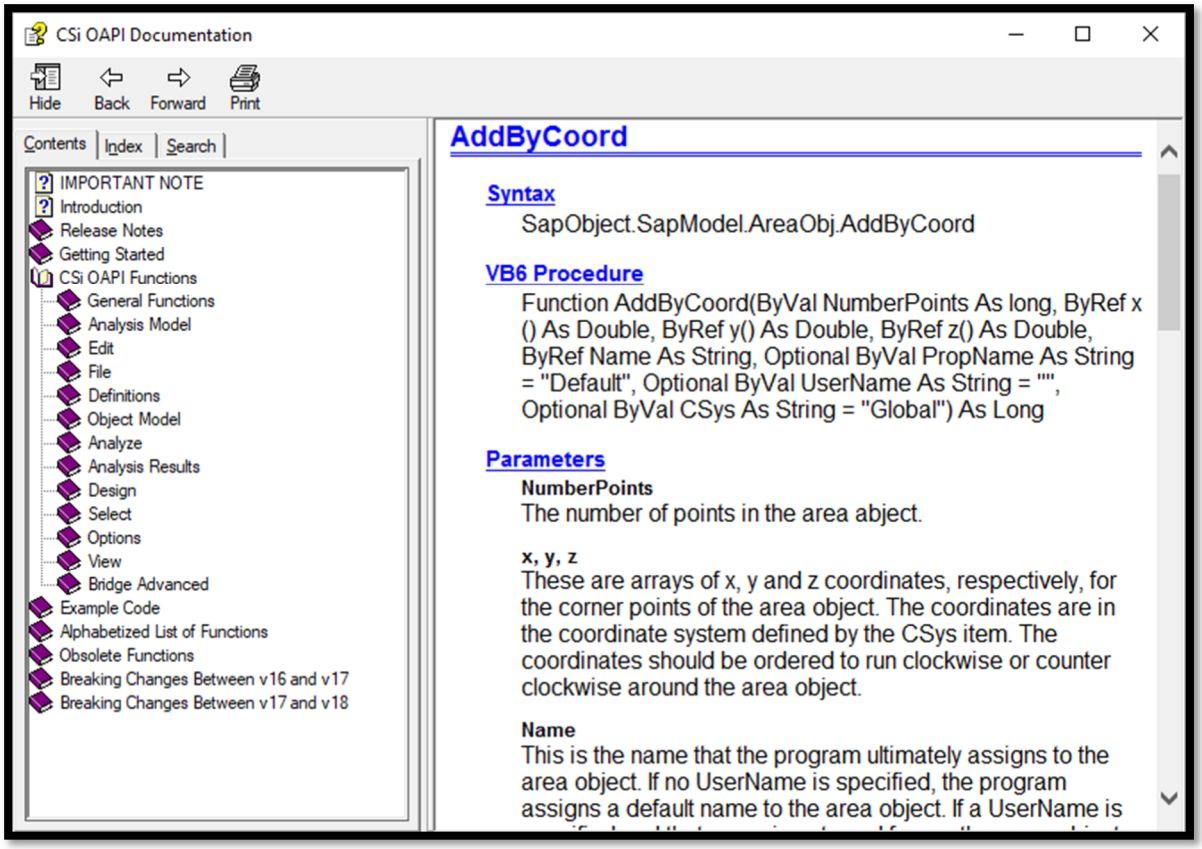


Figure 3: SAP2000 OAPI Documentation

Looking further into any of these sections, you will eventually find a list of methods to carry out commands in SAP2000 that can be used in an object-based programming language. Within the documentation for each of these methods, a few things are contained:

- Syntax for calling the method
- A list of the arguments (parameters) for the method, along with explanations
- Input/Output data types of the method
- Remarks about the use of the method
- An example of this method being used in VBA

An example of the method used to create a node with given cartesian coordinates is shown in Figure 4. In this method, the parameters “x”, “y”, and “z” are arguments of the “Double” data type. When using this method, the variables used to input “x”, “y”, and “z” must be declared as “Double” in order for the method to work. Another thing to note is that typically, parameters referenced as “ByVal” are input arguments that must be assigned values by the user. Those referenced as “ByRef” will be passed blank variables that are assigned values by SAP2000 and returned to the user after the method is carried out. It should be noted that some parameters are also referenced as “Optional”, which means that they do not have to be entered as an argument. If the variable is not entered when the method is called, the default value for that variable will be used for the method.

This is likely quite confusing to someone new to object-oriented programming and API’s. How methods are called in the SAP2000 API will become much more apparent after seeing examples.

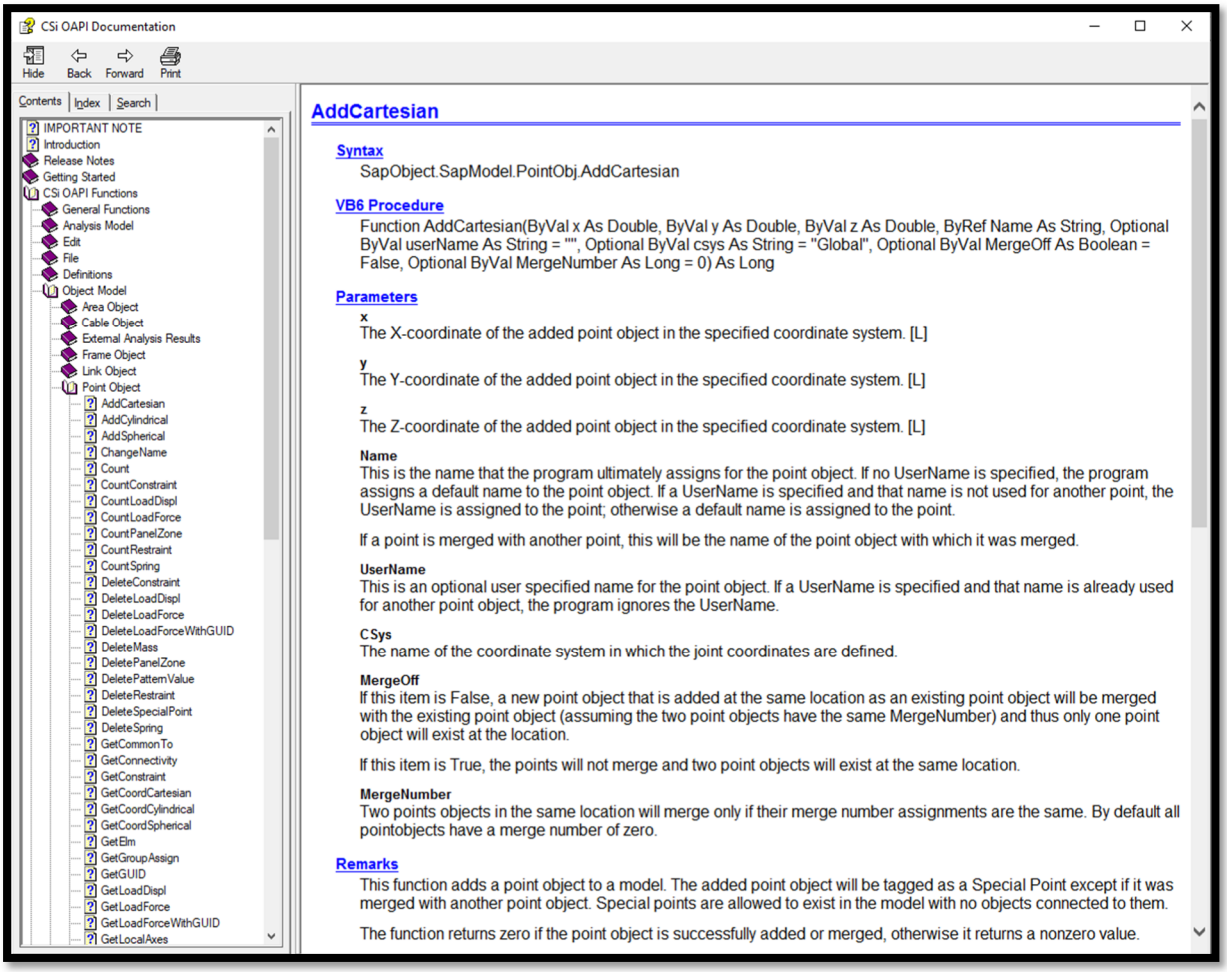


Figure 4: Method to Create a Node in SAP2000

3.4 – Examples

3.4-1 – Connecting an Instance of SAP2000 to a VBA Macro

In order to start using VBA macros to interact with an instance of SAP2000, the VBA macro must first be connected to that instance. In order to accomplish this, the user must open an instance of the SAP2000 application and also create a macro-enabled Excel file. Once in Excel, open the VBA editor window. Once in the editor, go to Tools > References in order to open up the references window. To use the methods and object types specified by SAP2000, the user must manually add the corresponding API library to their excel project. In the case of this example, the SAP2000 version is v20, so that is the library that will be selected, as shown in Figure 5. Press "OK" once the reference has been checked.

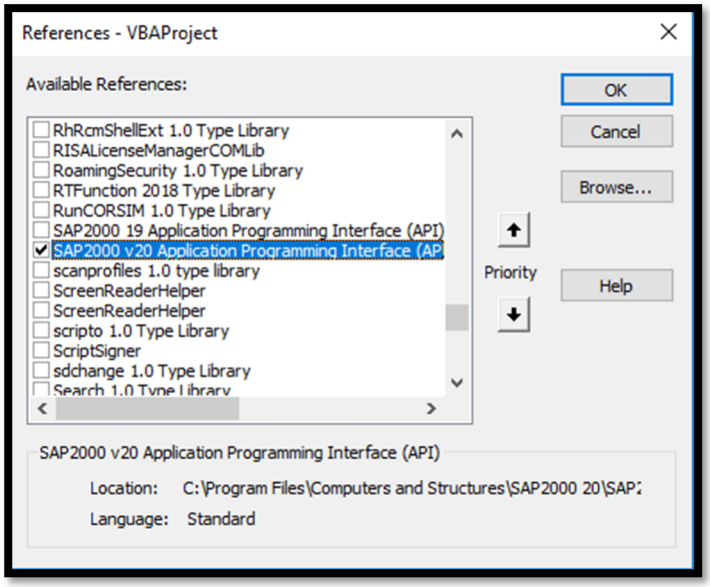


Figure 5: Adding the SAP2000 OAPI Reference

Next, insert a new module into your Excel project. Within this module, the following code can be added. This code is broken into multiple sections to be better understood by the user, although this is not necessary. The specifics of what exactly is happening behind the scenes here is not important for a new user. The basic idea is that a new variable called "SapModel" is being created that will reference the currently open model in SAP2000, and this variable is an object type specified by the SAP2000 API. Once this connection has been made, the external SAP2000 model essentially exists as a variable within the VBA code. Any method the user uses on this "SapModel" variable will execute the equivalent command in the real SAP2000 model. After the user's code has been executed, the "SapModel" variable is erased in order to conserve computer memory.

```
Sub API_Demo()

' dimension the sap object and sap model
Dim SapObject As cOAPI
Dim SapModel As cSapModel

' set the sap object as the current instance
Set SapObject = GetObject(, "CSI.SAP2000.API.SapObject")

' attach current sap model to the variable
Set SapModel = SapObject.SapModel

' --- START YOUR CODE HERE --- '



' --- END OF USER CODE --- '

' releases memory resources
Set SapModel = Nothing
Set SapObject = Nothing

End Sub
```

Code Block 9: Connecting an Instance of SAP2000

3.4-2 - Creating and Editing Geometry in SAP2000

Now that a SAP2000 model exists as a variable in the VBA macro, the possibilities are endless! The same code as in the previous example will be used for this example, and the actual code will be placed between the start and end points specified by the comments.

This example will start by creating some geometry in the SAP2000 model according to some simple parameters. It will be assumed that the reader has enough experience with SAP2000 to understand some of the basic terminology and editing/analysis features of the program throughout these examples.

To start off, go to Define > Section Properties > Frame Sections in SAP2000 and then select Import New Property. Import the W12X40 and W14X53 cross-sections. This is something that can be done with the API, but it is very simple to do manually and only needs to be done once. The rest of the example will be done completely in a VBA macro. The code is shown in Code Block 10.

```
Global n_bays As Integer
Global n_stories As Integer
Global width As Double
Global height As Double
Global Col_Size As String
Global Beam_Size As String

Sub API_Example_2()

' dimension the sap object and sap model
Dim SapObject As cOAPI
Dim SapModel As cSapModel

' set the sap object as the current instance
Set SapObject = GetObject(, "CSI.SAP2000.API.SapObject")

' attach current sap model to the variableS
Set SapModel = SapObject.SapModel

' --- START YOUR CODE HERE --- '
' unlock the model if it is already locked!
ret = SapModel.SetModelIsLocked(False)

' set units and select & delete existing framing
ret = SapModel.SetPresentUnits(eUnits_lb_ft_F)
ret = SapModel.SelectObj.All(False) ' selects all objects in model
ret = SapModel.FrameObj.Delete("ignore", eItemType.eItemType_SelectedObjects)
ret = SapModel.PointObj.DeleteSpecialPoint("ignore", eItemType_SelectedObjects)

' create parameters to characterize a moment frame
' these can be easily changed to affect the whole geometry
n_bays = 2
n_stories = 2
width = 20 ' 20 feet
height = 14 ' 14 feet
Col_Size = "W14X53"
Beam_Size = "W12X40"

' add points to the model, setting restraints to ground level nodes
Dim x As Double
Dim y As Double
Dim z As Double
Dim Name As String
Dim UserName As String
Dim Value(0 To 5) As Boolean
Value(0) = True ' U1 translation fixity (true = fixed)
Value(1) = True ' U2 translation fixity
Value(2) = True ' U3 translation fixity
Value(3) = True ' R1 rotation fixity
Value(4) = True ' R2 rotation fixity
Value(5) = True ' R3 rotation fixity
```

```
For b = 0 To n_stories
    For c = 0 To n_bays
        ' create nodes at each beam & column junction
        x = 0 + c * width
        y = 0
        z = 0 + b * height
        UserName = "Node_" + CStr(b) + "_" + CStr(c)
        ret = SapModel.PointObj.AddCartesian(x, y, z, Name, UserName)
        ' set full fixity if on base level
        If b = 0 Then
            ret = SapModel.PointObj.SetRestraint(Name, Value)
        End If
    Next c
Next b

' create a group in which all frame elements will be added to
ret = SapModel.GroupDef.SetGroup("Frames")

' add columns to the model by specifying start & end nodes
Dim Point1, Point2 As String
Dim PropName As String
PropName = Col_Size

For b = 0 To n_stories
    For c = 0 To n_bays
        Point1 = "Node_" + CStr(b) + "_" + CStr(c)
        Point2 = "Node_" + CStr(b + 1) + "_" + CStr(c)
        UserName = "Col_" + CStr(b) + "_" + CStr(c)
        ret = SapModel.FrameObj.AddByPoint(Point1, Point2, Name, PropName, UserName)
        ret = SapModel.FrameObj.SetGroupAssign(Name, "Frames")
    Next c
Next b

' add beams to the model by specifying start & end nodes
PropName = Beam_Size
For b = 1 To n_stories
    For c = 0 To n_bays - 1
        Point1 = "Node_" + CStr(b) + "_" + CStr(c)
        Point2 = "Node_" + CStr(b) + "_" + CStr(c + 1)
        UserName = "Beam_" + CStr(b) + "_" + CStr(c)
        ret = SapModel.FrameObj.AddByPoint(Point1, Point2, Name, PropName, UserName)
        ret = SapModel.FrameObj.SetGroupAssign(Name, "Frames")
    Next c
Next b

' --- END OF USER CODE --- '

' releases memory resources
Set SapModel = Nothing
Set SapObject = Nothing

End Sub
```

Code Block 10: Creation of Geometry Using SAP2000 API

At the very top (outside of the subroutine) several variables are dimensioned as global variables because these parameters will be used in other modules and subroutines in following examples. Values are assigned to these variables later in the code.

In the first section of code, the model is unlocked, the units of the SAP model are set, and all existing geometry in the model is selected and then deleted. The code will still execute without this part, especially for the first time the script is run (when no geometry exists in the model). However, deleting geometry at the beginning of the script using VBA prevents you from having to delete the geometry manually every time you run the script. If the geometry is not deleted manually or deleted with code, consecutive runs of the code will place multiple framing members and nodes on top of each other in the same location.

The second section assigns values to the parameters that characterize a moment frame; these can be easily changed by the user. For practical purposes, it would be useful to create these inputs in a spreadsheet and then import them to the script when it is run. This is not being done for clarity of the example. In this case, a two-bay and two-story moment frame will be created with the given dimensions. The beam and column sizes are also specified.

Next, the creation of geometry in the model starts. Nested "For" loops are utilized in order to create a series of joints using the PointObj.AddCartesian() method. Note that the variables are all dimensioned first with the specified data type listed in the API documentation. Once the relevant variables have been dimensioned, the looping can start. The loops are set up so that the joints will be created level-by-level, starting at the bottom; within each level, the code will work its way left-to-right. At each point in the loop, the coordinates are re-calculated, a clever user-specified name is set for each node (of the form "Node_*LevelNumber*_*ColumnNumber*") using the "UserName" variable. Then, if the newly created node is on the base level (b = 0), the restraints are set to fully fixed using the PointObj.SetRestraint() method. The way in which the method is called can be found at the top of the page in the documentation for that particular method under the "Syntax" section. Also, note that each method is called by setting the variable called "ret" equal to the method. This is a convention that SAP2000 uses. The "ret" variable will be set equal to zero if the method is successfully carried out. If it fails (which could be for a variety of reasons), then the value will be non-zero.

Subsequently, a group called "Frames" is created in the model that will be used to store all frame objects. In the next section, the columns are created with the FrameObj.AddByPoint() method using a similar nested loop structure as the previous section. For each column, the start and end point are stored as "Point1" and "Point2" variables, and a clever user-specified name is set for each column (of the form "Col_*BaseNodeLevel*_*ColumnNumber*"). This method also requires that a section property be specified; in this case the variable "PropName" is passed the column size specified at the beginning of the code. The next section follows a format very similar to this in order to create the beams. Each column and beam frame object that is created is added to the "Frames" group using the FrameObj.SetGroupAssign() method.

Once the macro has been created, run it! After it has been executed successfully, the geometry should be instantly created in SAP2000 and should look like Figure 6. The node and frame labels are shown in order to demonstrate that they have been successfully set to what the code specifies. The base nodes are also set to be fully fixed.

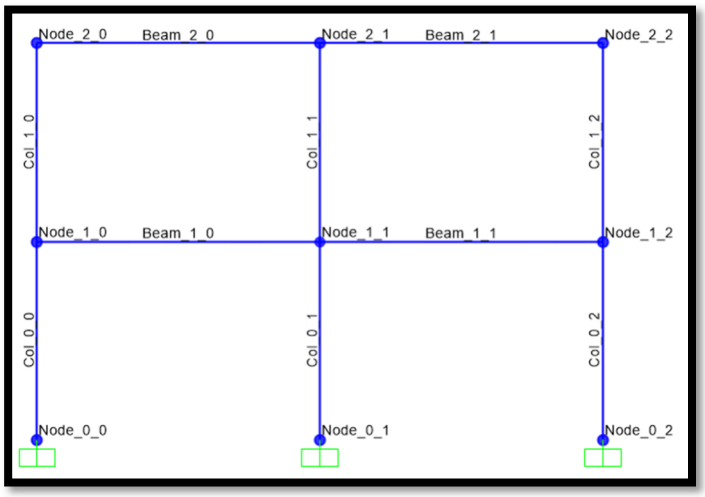


Figure 6: Result of Geometry Creation Code

Now, by changing the parameters specified at the beginning of the script, a large number of various moment frame configurations could be created with minimal time and effort. Try playing around with this to see for yourself.

3.4-3 - Creating Load Cases and Load Combinations

In this example, loads will be applied to the structure. To start, the load patterns, cases, and combination will be manually created within SAP2000. The reason for this is that these are simple to create and only need to be created once; if the names changed each time the macros were run, other considerations would need to be made. The load patterns, cases, and combination to be defined are shown in Figure 7.

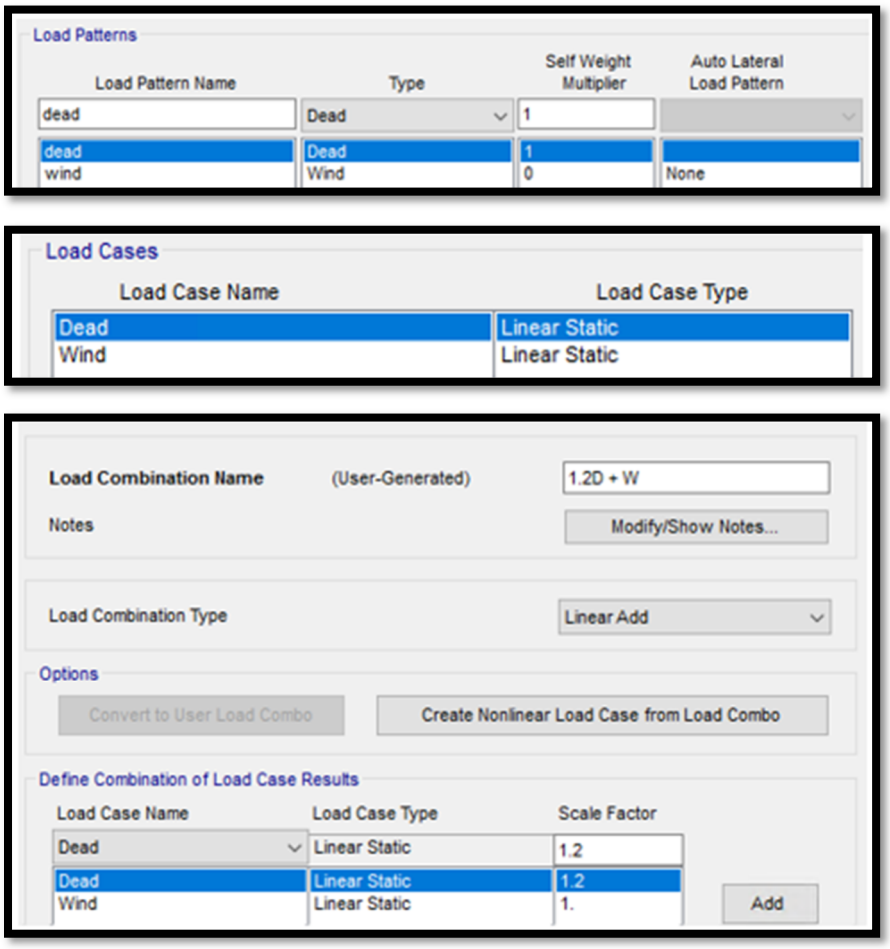


Figure 7: Manually Created Load Patterns, Cases, and Combo

Once the load patterns, cases, and combo have been created, the code can be written to assign actual loads to the model. This code is shown in Code Block 11. The first section of code creates variables for the user-specified load intensities for each load pattern. These variables will be used later.

Next, the process of applying distributed dead loads to the frames starts. This will be done for each frame using the FrameObj.SetLoadDistributed() method. Arguments for this method are declared in the first part of this section; variables such as “MyType”, “Dir”, “Dist1”, and “Dist2” might seem confusing until the documentation for this method is referenced. Comments are placed in the code in order to help make their purposes more apparent.

```
' --- START YOUR CODE HERE --- '
ret = SapModel.SetPresentUnits(eUnits_lb_ft_F)

' specify load intensities
Dim dead_load As Double
Dim wind_load As Double
dead_load = 50 ' 50 plf
wind_load = 50000 ' 50 kip point load

' declare variables relating to dead loads on frames
Dim Name As String
Dim LoadPat As String
Dim MyType As Double
Dim Dir As Long
Dim Dist1, Dist2 As Double
Dim Val1, Val2 As Double
MyType = 1 ' indicates Force/Length load type
Dir = 10 ' indicates gravity load direction
Dist1 = 0 ' start of distributed load at member start
Dist2 = 1 ' end of distributed load at member end
Val1 = dead_load ' load will be uniformly distributed
Val2 = dead_load

' add distributed dead loads to the beams
LoadPat = "dead"
For b = 1 To n_stories
    For c = 0 To n_bays - 1
        Name = "Beam_" + CStr(b) + "_" + CStr(c)
        ret = SapModel.FrameObj.SetLoadDistributed(Name, LoadPat, MyType, Dir, Dist1, Dist2, Val1, Val2)
    Next c
Next b

' declare variables relating to wind loads on nodes
Dim Value(0 To 5) As Double
Dim Replace As Boolean
Value(0) = wind_load ' Force in DOF 1
Value(1) = 0 ' Force in DOF 2
Value(2) = 0 ' Force in DOF 3
Value(3) = 0 ' Moment in DOF 4
Value(4) = 0 ' Moment in DOF 5
Value(5) = 0 ' Moment in DOF 6
Replace = True ' will replace the loads already applied to node

' add lateral point loads to the left-most nodes
LoadPat = "wind"
For b = 1 To n_stories
    Name = "Node_" + CStr(b) + "_0"
    ret = SapModel.PointObj.SetLoadForce(Name, LoadPat, Value, Replace)
Next b

' --- END OF USER CODE --- '
```

Code Block 11: Applying Loads Using the API

Once the variables are declared, the “dead_load” value assigned before will be applied to the magnitude of load at the start and end (“Val1” and “Val2”), creating a uniform load. Next, nested “For” loops are used to iterate through all of the beams in the model and apply the specified distributed load to each of them.

The last section of code applies wind point loads to each of the left-most nodes using the PointObj.SetLoadForce() method. Similarly to the distributed loads, relevant variables are declared and assigned values. A “For” loop is used to iterate through the left-most nodes in order to apply the point loads.

Displaying the “dead” and “wind” load patterns in SAP2000 after the macro has been run should yield the results shown in Figure 8.

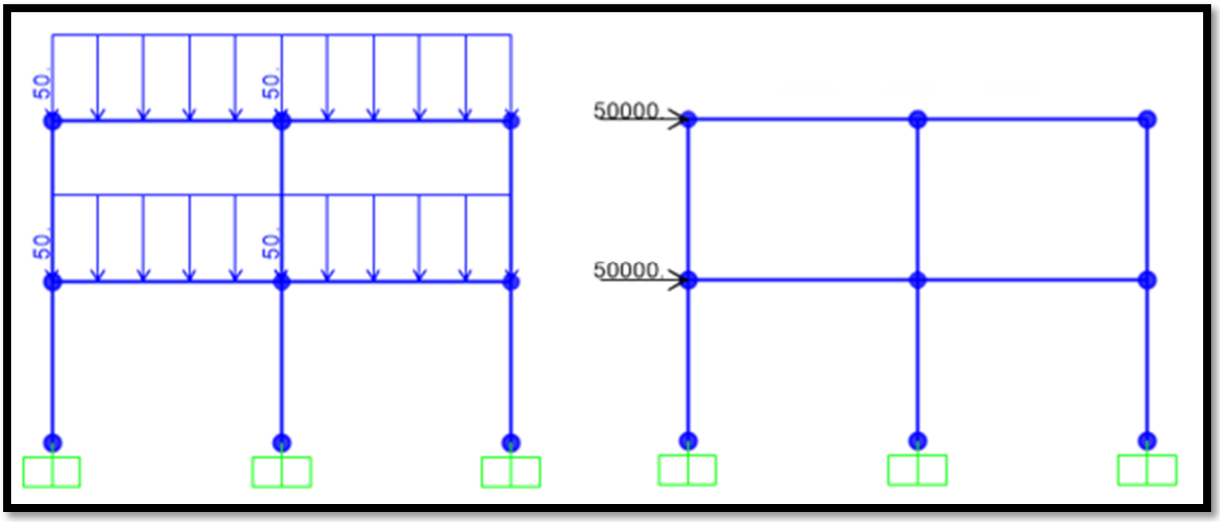


Figure 8: Dead and Wind Load Patterns after Load Application Macro

3.4-4 - Running an Analysis

Up to this point, the code developed in the examples will draw the geometry of a frame that is parameterized by the user and assign gravity and lateral loads to it. The next logical step in the process will be to run an analysis of the loads that have been assigned. As shown in Code Block 12, the process is as simple as unlocking the model and using the Analysis.RunAnalysis() method. Unlocking the model first is not necessary if the model is being run for the first time, but it is always good practice to include. If multiple analysis iterations are implemented, then unlocking the model will be necessary.

```
' --- START YOUR CODE HERE --- '
ret = SapModel.SetPresentUnits(eUnits_lb_ft_F)

' unlock the model if it is already locked
' analysis only runs when model is unlocked
ret = SapModel.SetModelIsLocked(False)

' run the analysis
ret = SapModel.Analyze.RunAnalysis()

' --- END OF USER CODE --- '
```

Code Block 12: Running the Analysis

The deformed shape that results after this code has been executed is shown in Figure 9.

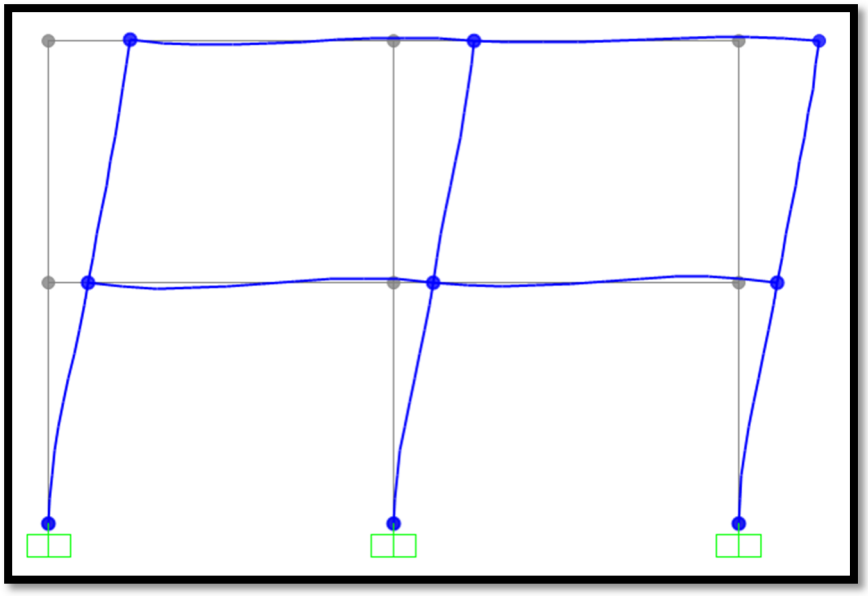


Figure 9: Deformed Shape of Combination 1.2D + W after Analysis

3.4-5 - Extracting Analysis Results

After running the analysis, results of that analysis will exist within SAP2000. In practice, the results are typically accessed and post-processed in one of two ways. One way is by inspecting the results visually through the SAP2000 GUI and doing subsequent calculations in other programs or by hand. This can be particularly grueling and time-consuming. Another way is by using the output tables in SAP2000, which can be accessed by Display > Show Tables. This will provide the specified analysis output in spreadsheet format. This can be much more useful, but the spreadsheet contains raw data that are not usually organized in a particularly useful configuration. Because of this, the raw data must be sifted through and re-organized, which can take a substantial amount of time (especially for very large projects). It should also be noted that this sifting procedure will have to be done repeatedly every time new analysis results are created.

A much more efficient and painless method of extracting pertinent data is to use the SAP2000 API. With this, it is possible to access any analysis results that you would be able to access within the GUI and more. Without the limiting factor of sifting through data and dealing with a GUI, the post-processing capabilities are only limited by the user's imagination.

In this example, base reactions and story drifts will be extracted from the analysis. This process is shown in Code Block 13. First, the units are set to kips and inches, which is useful for viewing forces and deflections. Next, the user must specify which cases or combinations they would like to receive results for. The best way to do this consists of two steps: first to deselect all case/combos, then to go back and select the cases/combos of interest. The deselection step will simply clear any cases or combos that were previously selected if there was code run before this point; it is typically good practice to include this. The method used for deselection is Results.Setup.DeselectAllCasesAndCombosForOutput. The method used for selection is Results.Setup.SetComboSelectedForOutput(). There is an equivalent method for selecting cases, rather than combos, if they are of interest, also.

```
' --- START YOUR CODE HERE --- '
' setting units to kips and inches
ret = SapModel.SetPresentUnits(eUnits_kip_in_F)

' first, deselect all cases and combos for output
ret = SapModel.Results.Setup.DeselectAllCasesAndCombosForOutput

' set the combo "1.2D + W" as the selected output combo
ret = SapModel.Results.Setup.SetComboSelectedForOutput("1.2D + W", True)

' get drift (joint displacement) at each right-most roof node
' then writing to spreadsheet table
Dim Name As String
Dim NumberResults As Long
Dim Obj() As String
Dim Elm() As String
Dim LoadCase() As String
Dim StepType() As String
Dim StepNum() As Double
Dim U1() As Double
Dim U2() As Double
Dim U3() As Double
Dim R1() As Double
Dim R2() As Double
Dim R3() As Double

Dim WS As Worksheet ' dimension the worksheet
Set WS = Worksheets("Sheet1")
Dim OutputTable As Range ' setting area in spreadsheet to print results to
Set OutputTable = WS.Range(WS.Cells(1, 1), WS.Cells(n_stories + 1, 2))
OutputTable.Clear ' clears current values
OutputTable(1, 1).Value = "Node Name" ' setting column headers
OutputTable(1, 2).Value = "Lateral Drift [in]"

Dim c As Integer
c = n_bays ' only interested in right-most column line
For b = 1 To n_stories
    Name = "Node_" + CStr(b) + "_" + CStr(c) ' node name for results
    ret = SapModel.Results.JointDispl(Name, 0, NumberResults, Obj, Elm, LoadCase, _
    StepType, StepNum, U1, U2, U3, R1, R2, R3) ' underscore is a line break for clarity
    OutputTable(1 + b, 1).Value = Name ' writing name to output table
    OutputTable(1 + b, 2).Value = U1(0) ' writing lateral node displacement to table
    OutputTable.Rows(1 + b).NumberFormat = "0.000" ' truncating decimals
Next b
' get base reactions of the ground level nodes
Dim F1() As Double
Dim F2() As Double
Dim F3() As Double
Dim M1() As Double
Dim M2() As Double
Dim M3() As Double

Dim OutputTable2 As Range ' setting area in spreadsheet to print results
Set OutputTable2 = Worksheets("Sheet1").Range(WS.Cells(1, 4), WS.Cells(n_bays + 1, 7))
OutputTable2.Clear ' clears current values
OutputTable2(1, 1).Value = "Node Name" ' setting column headers
OutputTable2(1, 2).Value = "Fx [kips]"
OutputTable2(1, 3).Value = "Fz [kips]"
OutputTable2(1, 4).Value = "M [kip-in]"

For c = 0 To n_bays
    Name = "Node_0_" + CStr(c) ' node name for results
    ret = SapModel.Results.JointReact(Name, 0, NumberResults, Obj, Elm, LoadCase, _
    StepType, StepNum, F1, F2, F3, M1, M2, M3) ' underscore is a line break for clarity
    OutputTable2(2 + c, 1).Value = Name ' writing name to output table
    OutputTable2(2 + c, 2).Value = F1(0) ' writing lateral force to table
    OutputTable2(2 + c, 3).Value = F3(0) ' writing vertical force to table
    OutputTable2(2 + c, 4).Value = M2(0) ' writing moment to table
    OutputTable2.Rows(2 + c).NumberFormat = "0.000" ' truncating decimals
Next c

' --- END OF USER CODE --- '
```

Code Block 13: Extracting Analysis Results with the API

Once the relevant cases and combinations have been selected for output, a range from the worksheet is stored in a variable called “OutputTable”, which will be used to write the results to and display the output. It is common to post-process the analysis results within the code itself, but for the purposes of this example, the results will simply be written to the worksheet for visual inspection.

Next, the data extraction process begins. The first set of results collected in the example are story drifts (in the form of node displacements) at each level. This is completed by iterating through each of the levels and using the right-most node at that level to gather deflections using the Results.JointDispl() method. These results are then written to the spreadsheet using the “OutputTable” variable. Many of the arguments for the joint displacement method are not relevant in this situation. For example, “NumberResults” is irrelevant because we are only gathering the displacement for a single static load case. Similarly, “StepType” and “StepNum” are only used in more complex analyses, like static pushover and time history. Even though they are not applicable for the selected load combination, they must still be entered as arguments into the method in order for it to be executed properly. Each of these arguments work slightly differently depending on the type of analysis that is performed; refer to the SAP2000 documentation for more information.

The second set of results collected are base reactions of the ground level nodes. The process for this is very similar to before. First, an output table is created (in this case, it is created in the same spreadsheet as the previous one). Then, a loop iterates through each of the base nodes from left to right. For each node, the Results.JointReact() method is used to extract the “F1” (lateral), “F3” (vertical), and “M2” (moment) node forces. These forces are written to the spreadsheet as well.

After the macro has been executed, the results will look like that of Figure 10. The base reactions and node displacement results are shown from the SAP2000 GUI as well for reference.

Extracting analysis results for nodes is one of the simplest and most useful ways to leverage the capabilities of the API. Once the data have been written to the spreadsheet, post-processing tools for designing foundations would be relatively easy to include, for example.

It should be noted that the analysis result extraction capabilities of the SAP2000 API go much further than what is shown in this example. Results for any member type (frames, shell elements, and even solids) can be easily gathered. Results for several load cases and combinations can also be gathered at once with ease. Creating your own macro with a similar script to the one shown in this example and fidgeting with the different analysis types is the best way to learn about this. It also helps to explore the returned variables and the format that they take in the Locals window in the Visual Basic editor.

	A	B	C	D	E	F	G
1	Node Name	Lateral Drift [in]		Node Name	Fx [kips]	Fz [kips]	M [kip-in]
2	Node_1_2	1.804		Node_0_0	-31.398	-25.058	-3700.007
3	Node_2_2	3.714		Node_0_1	-37.792	6.357	-4013.326
4				Node_0_2	-30.810	32.674	-3630.897

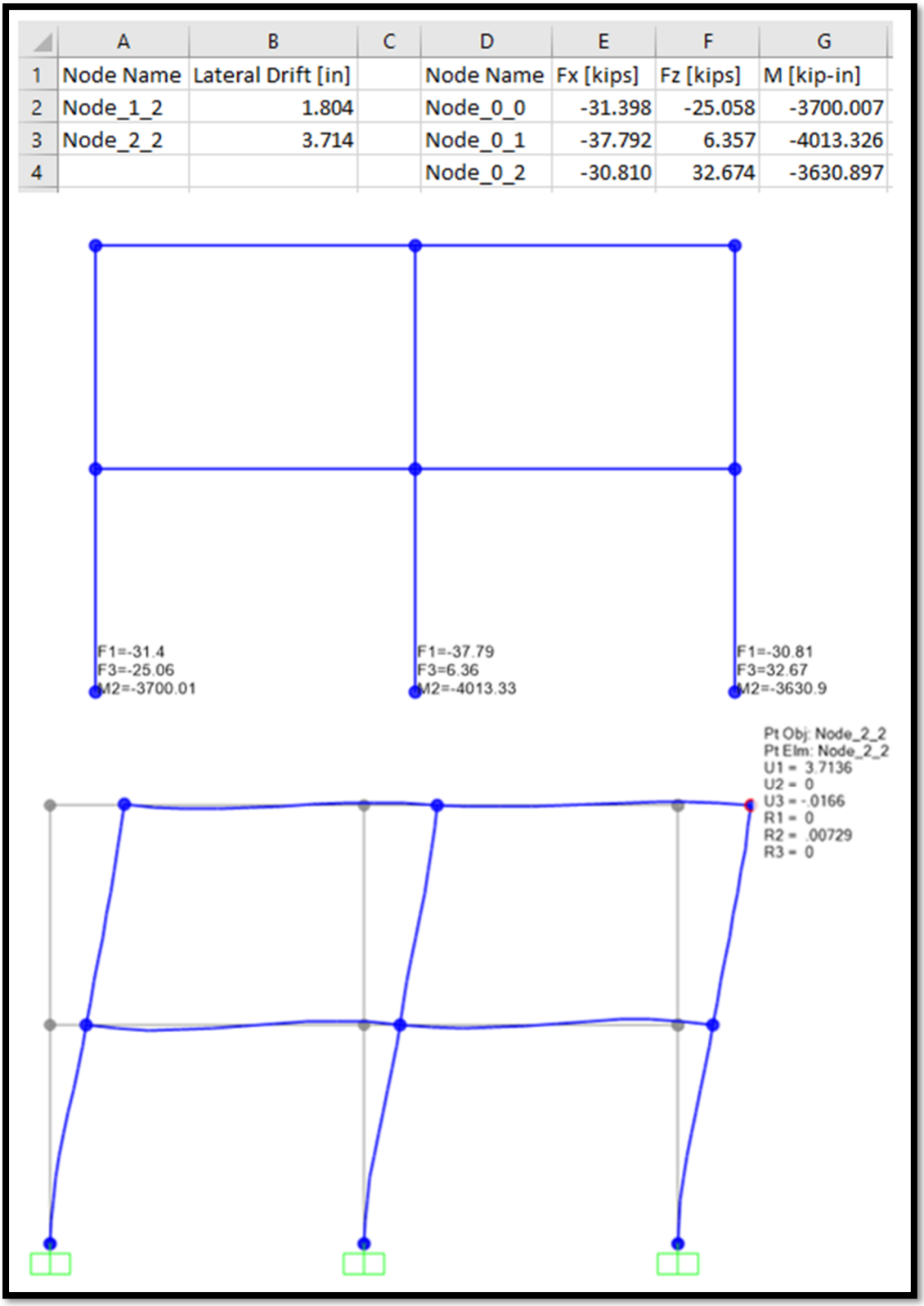


Figure 10: Spreadsheet Table after Extracting Analysis Results

3.4-6 - Running the Steel Design Module

This final example in the series will entail running the steel design module on the framing from previous examples and writing the interaction (demand-to-capacity) ratios to the spreadsheet. The code used to accomplish this is shown in Code Block 14.

The first to do will be to turn off the automatically-generated load combinations using the DesignSteel.SetComboAutoGenerate() method. Now, all of the standard AISC combos used for steel strength and deflection design will not be used (DSTL1, DSTL2, etc. in SAP2000). This allows the user to fully customize which combinations will be checked. In the case of this example, the previously created "1.2D + W" combination will be used for code strength checks. This is set up in the steel design preferences with the DesignSteel.SetComboStrength() method.

Next, the steel design code is set to AISC 360-10, and the steel design module in SAP2000 is executed with two simple lines of code.

Once the code check has been carried out, the API can be used to extract the results to VBA. Using the DesignSteel.GetSummaryResults() method, the user can extract design results for either a single member or an entire group of members. For this part of the example, the previously created frame group named "Frames" will be used. In order to indicate to the API that it should extract data for a group rather than an element, the optional "ItemType" argument must be passed the enumeration "eItemType_Group". After this method has been executed, the variables passed into the method with the "ByRef" notation in the API documentation will be populated with design results.

In the next section of code, an output table in the spreadsheet is defined and designated as a range object similar to the previous example. A "For" loop then iterates through each of the items in the results list and writes the name of the frame the results are for as well as the controlling interaction ratio. A screenshot of the worksheet after the macro has run is shown in Figure 11, as well as the steel code check heat map in SAP2000 for reference.

```
' --- START YOUR CODE HERE --- '

' turn off automatically generated combinations
ret = SapModel.DesignSteel.SetComboAutoGenerate(False)

' turn user-defined combination for strength design
ret = SapModel.DesignSteel.SetComboStrength("1.2D + W", True)

' set design code and execute the steel design module
ret = SapModel.DesignSteel.SetCode("AISC 360-10")
ret = SapModel.DesignSteel.StartDesign()

' get a summary of the results
Dim Name As String
Dim NumberItems As Long
Dim FrameName() As String
Dim Ratio() As Double
Dim RatioType() As Long
Dim Location() As Double
Dim ComboName() As String
Dim ErrorSummary() As String
Dim WarningSummary() As String
Dim ItemType As eItemType

Name = "Frames" ' name of the group of interest
ItemType = eItemType_Group ' indicating results for a group, not an object
ret = SapModel.DesignSteel.GetSummaryResults(Name, NumberItems, FrameName, _
Ratio, RatioType, Location, ComboName, ErrorSummary, WarningSummary, ItemType)

' creating an output table and writing the results
Dim WS As Worksheet ' dimension the worksheet
Set WS = Worksheets("Sheet1")
Dim OutputTable3 As Range ' setting area in spreadsheet to print results to
Set OutputTable3 = WS.Range(WS.Cells(1, 9), WS.Cells(NumberItems + 1, 10))
OutputTable3.Clear ' clears current values
OutputTable3(1, 1).Value = "Frame Name" ' setting column headers
OutputTable3(1, 2).Value = "DCR"

For i = 1 To NumberItems
    OutputTable3(i + 1, 1).Value = FrameName(i - 1)
    OutputTable3(i + 1, 2).Value = Ratio(i - 1)
    OutputTable3.Rows(i + 1).NumberFormat = "0.000" ' truncating decimals
Next i

' --- END OF USER CODE --- '
```

Code Block 14: Steel Design Code Check

I	J
Frame Name	DCR
Col_0_0	0.962
Col_0_1	1.032
Col_0_2	0.967
Col_1_0	0.355
Col_1_1	0.637
Col_1_2	0.381
Beam_1_0	0.945
Beam_1_1	0.932
Beam_2_0	0.694
Beam_2_1	0.597

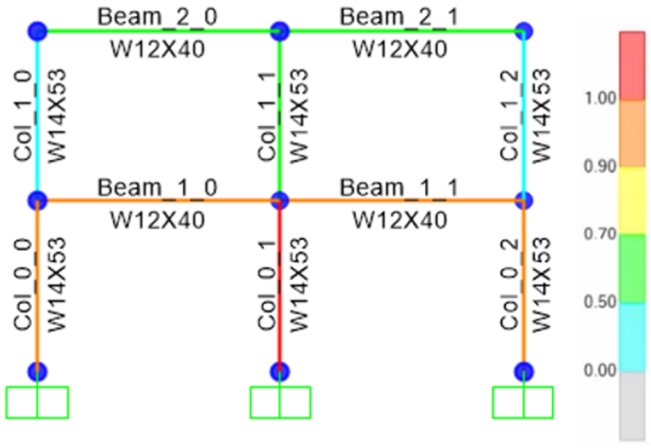


Figure 11: Steel Design Code Check Results

4 - Conclusion

Although most modern structural engineers have some knowledge of coding, it is typically limited to the realm of numerical analysis tools such as MATLAB. Having a familiarity with object-based programming can allow engineers to take their coding to a new level and interact with commercial structural engineering software in ways that can save an immense amount of time.

After seeing the simple examples of using VBA with the SAP2000 API, it is easy to imagine the feats that could be accomplished by expanding on these techniques. Automation of the geometry-building, load application, analysis, and result extraction phases of design can allow the user to make quick work of complex optimization problems. In the industry, it is also often used to carry out checks for discrepancies against the Building Information Modeling software and the analysis software being used on a project, which expediates the workflow and decreases the chances for human error to occur.

It is also important to reiterate that Visual Basic for Applications is not the only language that can be used with the SAP2000 API. In the documentation, examples are given for several languages including C#, C++, and even Python. Although VBA is a good choice when using excel to pre-process or post-process data, other languages may be superior depending on the situation.

Files containing the modules for all of these examples can be found on the linked Github page [4].

Appendix - Project-Specific Code Overview for Automation of SST Strong-Frame Connection Design

The Simpson Strong-Tie (SST) Strong Frame connection is a relatively new proprietary moment frame connection that can be found in the *AISC Prequalified Connections Specification for Seismic Applications,* referred to herein as the Prequalified Specifications [5]. This connection is unique in the fact that it uses external steel fuse elements connecting the beam flanges to the column to dissipate energy during a seismic event. Because of the way the fuse elements are designed, the plastic hinges will always form in these fuse elements, which are designed to be highly ductile and excellent at dissipating energy. Refer to Figure 12 for a schematic of the connection and its various components [5].

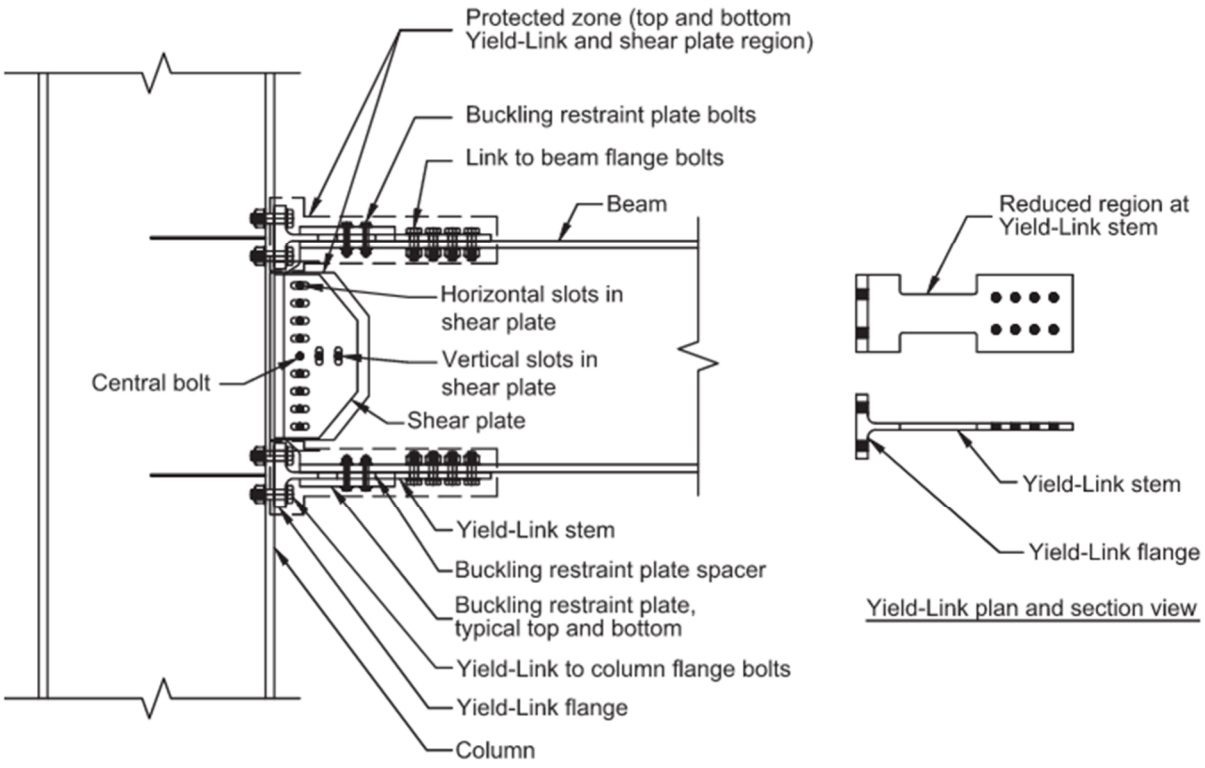


Figure 12: SST Strong Frame Connection Diagram

Although using this connection has many benefits, the design process is iterative and can be very tedious. Each iteration entails numerous calculations that result in small changes to be made to the geometry of the frame of interest in commercial design software. If the frame being designed is large, editing geometry can take a substantial amount of time because the connection components are modeled explicitly. Additionally, many iterations must be performed before a viable design for the connection is found. Because of this, a companion tool in Visual Basic for Applications was created to be used in concert with a spreadsheet for the connection design that followed the steps in the Prequalified Specification.

The design of a Strong Frame connection for a specific moment frame layout consists of three distinct steps, each requiring its own model in commercial structural analysis software. The first of these three steps is a “Pre-Fuse” analysis, where the structure is modeled as a typical moment frame with fixed end

connections; this is primarily used to get an idea of whether the trial sizes for the beams and columns will be viable, as well as finding an estimation of the fundamental period of the structure. The second step is an analysis of a frame beam on each level as a simply-supported member. This checks the viability of the frame beams for strength and end rotation limits. The final (and most complex) step is an analysis on the frame as a whole, where the Strong Frame connections being designed are modeled explicitly in the model. This step would take an immense amount of time without the help of tools that can automate the modeling and analysis process.

In the following sections, a summary of the methodology used for each of these design steps will be presented. The main design calculations are completed by the spreadsheet, and the VBA macros implement the results of the calculations in the SAP2000 model. Both the spreadsheet and accompanying SAP2000 file can be found in the linked Github page [6].

1 - Pre-Fuse Model

The first step begins by extracting the user-specified geometry from the spreadsheet. This includes member sizes, story heights, bay widths, loads, and bracing points. Once the macro has been connected to the open SAP2000 instance, all current geometry in the model is deleted.

Once the model is cleared, the creation of the geometry specified by the user can begin. Initial frame sizes are arbitrarily selected by the user and iteratively edited as the design process progresses. This starts by placing nodes where all of the beam and column ends will be. All elements created in the model are named with a convention that includes the level number and column line number; the level numbers start at zero at the base, and the column line numbers start from zero at the left. See Figure 13 for this convention on a small example model. The newly created nodes follow this convention. As the nodes at the base are created, their restraints are set to fully fixed.

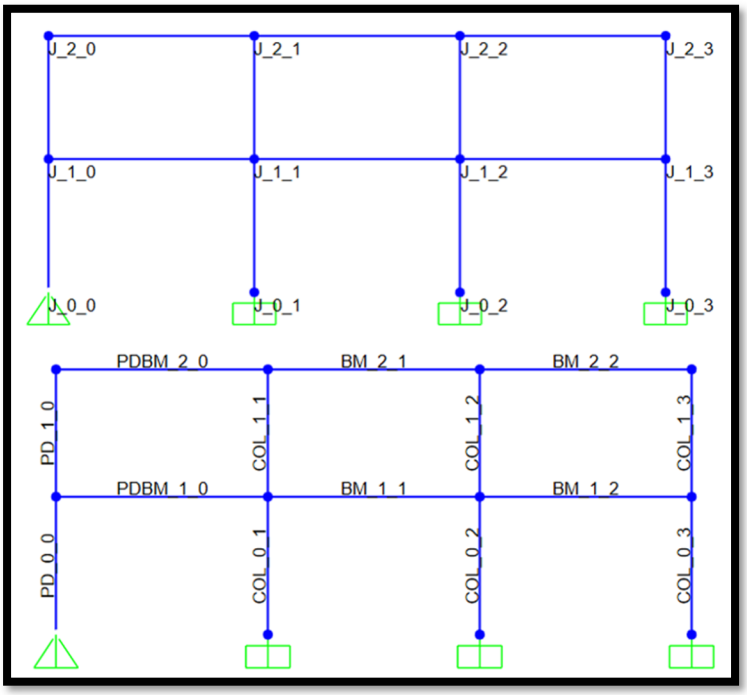


Figure 13: Pre-Fuse Model Naming Convention Example

The next part of the macro will create all of the columns in the model. The method used to create them requires arguments for both the start and end nodes. The start node is always the lower node. The nodes can be accessed easily due to the aforementioned standard naming convention. Each column is named with a similar naming convention, where the level number corresponds to the level of the bottom node.

The beam elements are created next; the same method used to create the columns are used for the beams. The beams are also created with this naming convention, where the column line corresponds to the column line of the left-most node. As the beams are created, bracing points are assigned according to the user input.

Next, all P-Delta beams and columns that have been entered into the model have their rotational end fixities released.

The next section of the macro begins the load application process. The distributed loads are applied to the beams first; this includes cladding dead load as well as additional dead and live loads. Gravity point loads are applied to the nodes of the exterior frame columns as well for the same loads previous mentioned. Gravity point loads are applied to the P-Delta columns in the same fashion. Next, masses and seismic lateral loads are added to the points on the frame. The seismic masses and lateral loads for each level are distributed evenly as point loads across the width of the frame. The seismic point loads are applied for all three seismic load cases: EX, EX_drift_nof, and EX_drift_fuse. These correspond to the loads generated from the code-calculated period, directly calculated period from the model without fuses, and the directly calculated period from the model with fuses.

After the geometry has been created, the analysis stage can be carried out. This process begins by running a modal analysis on the model to calculate the fundamental period. This period is then written to the spreadsheet, which will update the calculated seismic loads (EX_drift_nof). These newly calculated loads are re-imported into VBA and updated in the model; a subsequent analysis is then performed. Following this analysis, drift results are gathered for each of the levels and multiplied by 1.2 per the Prequalified Specification. The maximum moments at the face of the columns for the exterior and interior beams at each level are extracted from the analysis as well and written to the spreadsheet. This will be used for preliminary link sizing.

2 - Simply-Supported Beam Model

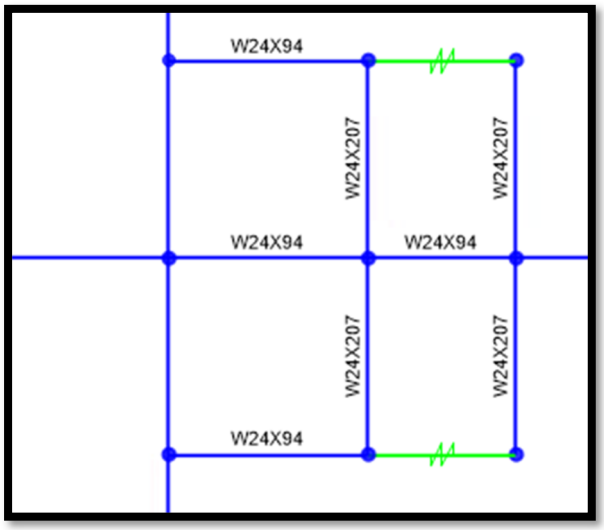
The second step is the simplest of the three, and only consists of analyzing a beam at each level as simply supported. Using a similar methodology as the first step, the existing geometry in the SAP2000 model is deleted and replaced with a simply-supported beam is created at each level. In each beam, a node is placed in the center in order to get exact displacements. The gravity loads from the spreadsheet are placed on each of these beams.

Next, the model is analyzed and a steel design code check is executed. The strength check interaction ratios for each beam are written to the spreadsheet as well as the service level deflection at each level.

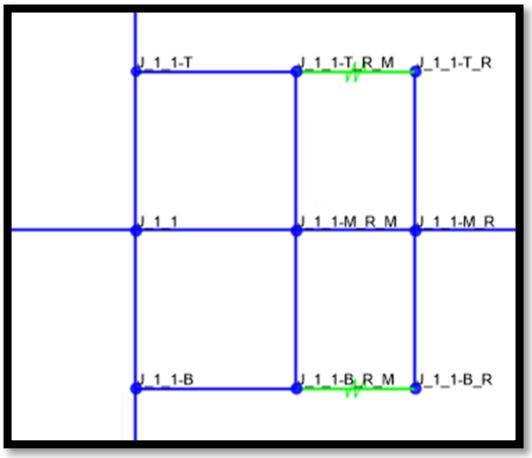
3 - Fuse Model

The final step of the design is the most complex. The geometry creation uses the same macro for the pre-fuse geometry to create the initial layout of the frame without any fuses. Next, a function is used that will draw the complex fuse geometry in at each beam-column join location.

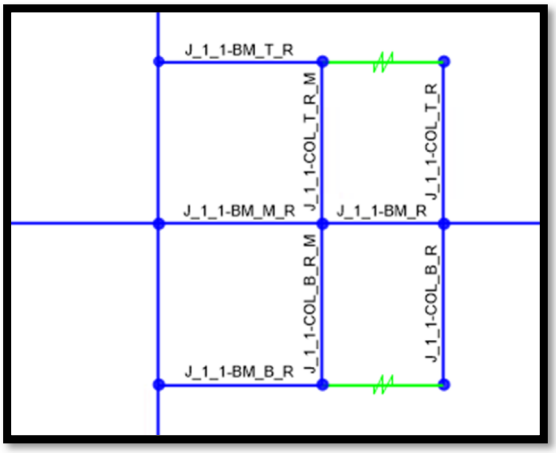
This function is fairly simple, but it is quite long considering how many elements must be created at each fuse connection location. The fuse connection geometry includes extra nodes, vertical and horizontal framing elements, and also link elements that contain the equivalent spring properties calculated per the Prequalified Specification. An example of one of the fuse connections modeled in SAP2000 is shown in Figure 14. The joint and frame labels are shown for reference. The framing element sizes match the surrounding beams and columns. Also, notice that all elements begin with the name of the host node at the beam-column join ("J_1_1" in this case) and contain an "R" to indicate that this connection is on the right side of the column.



(a) – Frame Element Sizes and Layout



(b) – Node Labels



(c) – Frame Element Labels

Figure 14: Layout and Naming Convention Example for a Fuse Connection Model

The analysis and post-processing of data for this step is a lengthy process, as the fuse model analysis is the last design step and must be used to confirm that the design is sufficient. The first step is similar to the pre-fuse model analysis. First, a modal analysis is carried out and the period (accounting for the fuses now) is extracted and written to the worksheet. With the newly calculated period, seismic loads are recalculated (EX_drift_fuse) and then re-applied to the SAP2000 model. With these new seismic loads calculated, the analysis is run again. Drifts at each level are recorded from the seismic lateral loads and written to the spreadsheet.

Next, the nonlinear envelope load case (ENVU_NL) is used to calculate the moments at interior and exterior beam ends within the links for each level. This is accomplished using the section cut functionality in SAP2000. During the creation of the connections, each of the link pairs are added to their own element group and corresponding section cut. During the analysis, the internal forces and moments are summed for each section cut. These internal moments can be reported as the moment in the fuse connections. The maximum moments at the interior and exterior connection types are found at each level and printed to the spreadsheet.

Next, the compression and tension forces are checked for the beams at every level with the load combination envelope ENVU_amp_NL. For each level, the maximum compression and tension forces are stored for each beam; a distinction is also made here between interior and exterior beams. If none of the members in the given level are controlled by tension (or compression) for any of the load combinations in the envelope, then the corresponding force value in the spreadsheet will be left blank and excluded from calculations. For the same check in the worksheet, the design capacities of all of the beams in question are also gathered from SAP2000 and written to the spreadsheet.

For the next section of this step, a similar process is carried out with columns. The design column capacities are extracted from SAP2000 and written to the spreadsheet. Then, depending on whether the user has selected ENVU_amp_NL or ENVU_NL as their chosen envelope, the macro will gather the maximum interaction ratios for the interior and exterior columns at each level and write it to the spreadsheet. This corresponds to “Option 2” of step 13.2 in the Prequalified Specification design process; “Option 1” is never used by the macro. The maximum shear and P-M ratio will be returned to the spreadsheet for each column type (exterior vs. interior) at each level.

Following this, the macro gathers column axial forces above and below each of the connections in order to carry out calculations for the beam-column relationship using the ENVU_amp_NL load combination envelope. All capacity calculations are completed externally in the spreadsheet.

The final section of code extracts the column axial forces below each of the connections from the ENVU_NL load combination envelope. These values are used for panel zone checks specified by the Prequalified Specification. This concludes the analysis process.

References

1. *Microsoft Excel*, Office 365, Microsoft, www.microsoft.com/en-us/microsoft-365/excel.
2. "Excel Visual Basic for Applications (VBA) Reference." *Excel Visual Basic for Applications (VBA) Reference | Microsoft Docs*, docs.microsoft.com/en-us/office/vba/api/overview/excel.
3. "Sap2000: Integrated Structural Analysis & Design Software." *Computers & Structures, Inc. - Structural and Earthquake Engineering Software*, v20, www.csiamerica.com/products/sap2000.
4. Dundore, Alex. "Files for SAP2000 Tutorial with VBA." *GitHub*, 11 Mar. 2021, https://github.com/alexdundore/SAP2000-VBA_Tutorial
5. Connection Prequalification Review Panel of AISC. *Prequalified Connections for Special and Intermediate Steel Moment Frames for Seismic Applications, Including Supplements No.1 and No.2*. 2020.
6. Dundore, Alex. "Simpson Strong Tie Strong Frame design spreadsheet and accompanying SAP2000 file." *GitHub*, 30 Mar. 2021, https://github.com/alexdundore/SST-SF_Design_Tool