Automating PrecisionTree with VBA

The purpose of this document is to introduce PrecisionTree’s Excel Developer Kit (XDK) and explain how you can use VBA to automate PrecisionTree. The term “automate” simply means that you write a program in VBA to make PrecisionTree do what you normally do through the PrecisionTree user interface. For example, suppose you often need to build a decision tree with a particular structure and then perform sensitivity analysis on its key inputs. This requires a considerable number of steps—virtually the same steps—each time you do it. Wouldn’t it be nice to click a button and have all of these steps magically happen? This is exactly the type of thing you will learn how to do with the PrecisionTree XDK.

Introduction to VBA¹

Visual Basic for Applications, or VBA, is the programming language for Excel. It is also the programming language for other applications in Microsoft Office: Word, PowerPoint, and Outlook, for example. It is even the programming language for some non-Microsoft products. To understand how this works, it helps to separate the name VBA into two parts: “Visual Basic” and “for Applications.” You can think of Visual Basic, or VB, as the “backbone” programming language. It contains programming elements that all programmers use, regardless of the programming language or the type of development task. For example, it contains subroutines, variable types, logical constructs such as If-Then-Else, loops for repetitive tasks, arrays for storing lists, and much more. Actually, there are non-VBA versions of VB. For example, you might be familiar with VB6 or VB.NET. All of these versions of VB, including VBA, are extremely popular because VB is fairly easy to learn and quite powerful. In any case, to learn VBA for Excel or PrecisionTree, you must first learn the backbone VB language.

The following is a typical segment of VB code. Even if you know nothing about programming, you will probably have little trouble understanding what this code accomplishes. Again, this is the attraction of VB. It is relatively easy to learn and understand. Note the lines that begin with an apostrophe. These lines are called comments. They are ignored by VBA when the program runs, but they are very useful for documentation.

```
Sub Invoice()
' Declare variables.
Dim nProducts As Integer, i As Integer
Dim total As Currency, subTotal As Currency
Dim nPurchased As Variant, unitPrice As Variant

' Define constants for this business.
Const taxRate = 0.06
Const cutoff1 = 50, cutoff2 = 100
Const discount1 = 0.05, discount2 = 0.1

' Enter information about this order
nPurchased = Array(5, 2, 1, 6)
unitPrice = Array(20, 10, 50, 30)

total = 0
' Loop over all products purchased.
For i = 1 To nProducts
  ' Calculate revenue, including possible discount, for this product.
  ...
```

¹ If you are already familiar with VBA for Excel, you can skip to the section “Introduction to the PrecisionTree XDK” on page 5.
subTotal = nPurchased(i) * unitPrice(i)
If subTotal >= cutoff2 Then
    subTotal = (1 - discount2) * subTotal
ElseIf subTotal >= cutoff1 Then
    subTotal = (1 - discount1) * subTotal
End If
' Add to total for this order.
total = total + subTotal
Next
' Add tax.
total = (1 + taxRate) * total
' Display result.
MsgBox "The total for this order, including tax, is " & Format(total, "$#,###00.00")
End Sub

If you run this code in Excel, you will see the display (a message box) in Figure 1.

Figure 1 Result of VBA Program

Subroutines, Programs, and Macros

Before proceeding, it is useful to discuss three terms you often hear: subroutine, program, and macro. A subroutine is any section of code that begins with the keyword Sub and ends with the keyword End Sub. Its purpose is typically to perform one specific task. A program is a collection of one or more related subroutines that achieves an overall goal. In practice, long programs are often broken up into smaller subroutines for enhanced readability and easier debugging. A macro is essentially the same as a program, and it is the term favored by many Excel programmers. However, if your programs are relatively short, consisting of a single subroutine, the terms subroutine, program, and macro are practically synonymous and are often used interchangeably.

Object Models

The above Invoice subroutine can be run in Excel, but it really has nothing to do with Excel. There are no references to ranges, formulas, worksheets, charts, or other items you use in Excel. The code is pure VB. But now we come to the “for Applications” part of VBA. All of the familiar items in Excel, which will now be called objects, are part of an object model that Excel provides to programmers through VBA. This means that you as a programmer can reference these objects in your VBA code. In short, this object model allows you to “make things happen” through VBA code, rather than through the usual Excel user interface. In this way, you can automate Excel with VBA.

So what does an object model, or Excel’s object model in particular, look like? At the top level, it is simply a list of things—objects—that the software contains. Excel’s object model is quite large because everything you see in Excel is an object. One of the most common objects is the Range object: any range of cells (including a single cell). But there are over a hundred more objects, including the Worksheet object, the Chart object, the PivotTable object, and so on. There are also singular and plural objects. For
example, there is the plural Worksheets object, the collection of all worksheets in an Excel workbook, and there is the singular Worksheet object, a particular worksheet in the workbook.

Figure 2 displays a partial list of the objects in Excel, plural (yellow) and singular (blue).

**Figure 2 Excel Object Model**

However, there is much more to an object model than the list of its objects. First, there is a **hierarchy** among many objects. Basically, this means that you often need to “drill down” to get to the object you want. A good example is a Range object, such as cell B3. The idea is that a range is contained in a worksheet, which is contained in a workbook. So the hierarchy is Workbook→Worksheet→Range, and the following line of code is typical:

```python
range = workbook.worksheets[0].range('B3')
```
You read this as: Starting in the active workbook, go to cell B3 of the worksheet named Costs, and enter the value 10. This concept of hierarchy is crucial to learning VBA for Excel or PrecisionTree.

Second, most objects have **properties** and **methods**. (Methods are also called **functions**.) If you think of objects as nouns, then you can think of properties as adjectives and methods as verbs. Properties describe an object, and methods indicate what you can do to, or with, an object. For example, a Range object has a Value property, which was used in the above line of code. Similarly, a Font object has a Color property, a Size property, and many others. A typical object with methods is the Worksheet object. It has a Delete method, a Move method, and a few others.

Although less common, many objects also have **events** that they respond to. A good example is the Open event of a Workbook object. This event occurs—it “fires”—when you open the workbook. VBA allows you to write **event handlers** for the event. This is code that is executed when the event fires.

As you begin VBA programming for Excel, you gradually learn the most common objects, the hierarchies between them, their most common properties, methods, and events, and the syntax for using these in VBA code. It takes practice and perseverance, but if you want to make Excel “sing and dance” with the click of a button, this is time well spent.

By now, you should be starting to see the big VBA picture. Just as the Excel application can be automated through its object model, other applications can be automated through their object models. Microsoft Word is a typical example. Its objects are not the same as Excel’s. Instead of Worksheet, Range, and other typical Excel objects, Word has Sentence, Paragraph, and other objects that are useful in word processing, and these objects have their own hierarchies, properties, methods, and events. Fortunately, if you already know VBA for Excel and you want to learn VBA for Word, all you need to learn is the object model for Word. The backbone VB language is exactly the same in both applications. Admittedly, it is not a trivial task to learn a new application’s object model, but knowing VB for one application, like Excel, provides a big head start for learning VB for another application, like Word.

**Using the Visual Basic Editor (VBE)**

You will be doing all of your programming in the Visual Basic Editor (VBE). The easiest way is to open VBE from Excel is to press **Alt+F11**. Alternatively, if the Excel Developer tab is visible, you can click the **Visual Basic** button on its ribbon. The VBE window appears in Figure 3. In particular, the Project pane on the left shows a list of all open workbooks. (To get back to the Excel window, you can click the Excel icon on the toolbar below the menu bar.)
As you can see in the figure, the selected file contains a module. You will write all of your code in modules. (Actually, there are exceptions to this, but they aren’t discussed here.) By default, a file doesn’t contain a module, but you can insert one through the Insert menu. Then you will see a big white space on the right where you can start typing your code. Some of the code for this particular module appears in the figure.

**Introduction to the PrecisionTree XDK**

Like Excel and Word, PrecisionTree has its own object model. It can be used to change application settings, build a decision tree, generate reports, run sensitivity analyses, and other PrecisionTree tasks. The details of these possibilities are discussed in some detail later in this document and in the accompanying PrecisionTree XDK example files.

Two questions probably come to mind. Do you really need to learn how to automate PrecisionTree with VBA, and, if you do, how do you learn the language?

There are at least two reasons why you might want to automate PrecisionTree with VBA. First, it lets you automate tasks that you perform repeatedly. For example, suppose you need to build a reasonably large decision tree with a particular structure, generate reports, and perform sensitivity analyses. If you do this once or twice, it is probably not worth the effort to write a VBA program to automate the process. However, if you do this repeatedly, a VBA program can replace many mouse clicks with a single click. Indeed, this is why VBA for Excel is so popular among users—it lets them automate repetitive tasks with the click of a button. This same motivation certainly applies to repetitive tasks in PrecisionTree.
Second, suppose you are developing models for colleagues or clients who have little experience with PrecisionTree. Instead of teaching them how the PrecisionTree user interface works, it might be easier to develop a VBA application that allows them to simply click a button to make implement the entire process.

The PrecisionTree object model has a fairly large number of objects. Many of the objects, including those you will use most often, appear in Figure 4. Specifically, most of the objects (also called “classes”) at the bottom right in this figure are associated with decision trees, the main focus of this guide and the PrecisionTree XDK example files. As you can see, there is also a section of the XDK for automating influence diagrams, but because influence diagrams are not used nearly as frequently as decision trees, this part of the XDK is not discussed here.

**Figure 4 PrecisionTree Object Model**

Assuming you think VBA for PrecisionTree is for you, how do you learn the language? First, you need some facility in VBA for Excel. As discussed earlier, this requires you to learn the fundamentals of the VB “backbone” language and the basics of the Excel object model. There are a number of reference books on VBA for Excel, as well as Excel’s own online help. One recommendation is the first 10 chapters of *VBA for Modelers* by Albright. This book provides concise explanations and plenty of example code to get you up to speed in both the VB language and the Excel object model.

Once you are familiar with VBA for Excel, you have at least four options—which are not mutually exclusive—for extending your knowledge to VBA for PrecisionTree.
1. You can continue reading this document. It provides an overview of what can be done, and it provides sample code for how to do it. Alternatively, you can watch the introductory video that covers much of the same material. It is available from the Developer Kit (XDK) item under PrecisionTree Help.

2. You can do what all good programmers do—mimic code written by others. Examples of programs written for PrecisionTree are provided in a number of PrecisionTree XDK example files. (They can be found from the Developer Kit (XDK) item under PrecisionTree Help.) Once you see how something is done, such as building a decision tree or performing sensitivity analysis, you will probably have little difficulty adapting the code for your own use.

3. You can open the Reference document from the Developer Kit (XDK) item under PrecisionTree Help and start browsing. This document is quite long, and no sane person would ever read it from beginning to end, but it gives you a quick sense of the PrecisionTree object model, and it often enables you to find the specific details you need for your program.

4. You can open the Object Browser that is part of the Visual Basic Editor. This is particularly useful if you are in the midst of writing a program and need information on one specific detail. The Object Browser is discussed later in this document.

By the way, if you are familiar with VBA for Excel, you might want to start by recording macros for PrecisionTree, just as you can do for Excel. Unfortunately, this doesn’t work. If you turn on the recorder and start clicking PrecisionTree buttons, you will get some recorded code, but it will be entirely irrelevant. To automate PrecisionTree, you have to write the code.

Setting Library References

Before you can access the PrecisionTree XDK with VBA code, you must first set references to two PrecisionTree libraries (from References in the Tools menu of the Visual Basic Editor). These are the \texttt{PtreeXLA} and \texttt{Palisade PrecisionTree 6.x Object Library} references, illustrated in Figure 5. (The former references the Ptree.xla add-in file. The latter is abbreviated as \texttt{PtreeOL6} in the Visual Basic Object Browser.) This should be the first step before writing any VBA macros to automate PrecisionTree.
Figure 5 PrecisionTree References

PrecisionTree Automation Guidelines

This section provides some guidelines for automating PrecisionTree with VBA code. The guidelines in this section are purposely kept brief and are intended only to give you the “lay of the land” and alert you to a few technical issues. For a more pictorial set of guidelines for the PrecisionTree object model, you should examine the file PrecisionTree XDK - Object Model Diagrams.xlsx. To see complete applications of PrecisionTree automation, you should look at the accompanying PrecisionTree XDK example files. And finally, you can always visit the PrecisionTree XDK documentation or the Object Browser in the Visual Basic Editor.

By the way, you will notice that all of the formal names of PrecisionTree objects start with PT, as in PTApplicationSettings. However, they are typically referenced by a property or method that doesn’t include the PT prefix, such as ApplicationSettings.

Getting Started: The “Root” Object

The “root” object that everything starts from is the PrecisionTree object, a reference to PrecisionTree. All other PrecisionTree objects are down the hierarchy from it. For example, you will see PrecisionTree.ModelWorkbook, PrecisionTree.ApplicationSettings, and others. These and other objects down the hierarchy will be discussed shortly.

Technical Note: PrecisionTree Function versus PrecisionTree Object

There is a subtle issue you should be aware of. As it is used here, PrecisionTree is a function in the PtreeXLA library that returns a PrecisionTree object in the PtreeOL6 library.2 To put it another way, if

---

2 Remember that PtreeOL6 is the abbreviation for the Palisade PrecisionTree 6.x Object Library.
you did not reference the PtreeXLA library and wrote a typical line such as the following, PrecisionTree wouldn’t be recognized.

```
PrecisionTree.ApplicationSettings.ReportPlacement = PTActiveWorkbook
```

However, if you do reference PtreeXLA, as you should, you can simply write PrecisionTree, followed by any of its properties or methods, and it will work fine. (In particular, you do not need to declare PrecisionTree as an object variable.)

**Changing Application Settings**

You can change application settings with `PrecisionTree.ApplicationSettings`, just as you can through the usual Utilities menu in PrecisionTree. For example, the following lines change the report placement setting and guarantee that new reports won’t overwrite existing reports.

```
With PrecisionTree.ApplicationSettings
    .ReportPlacement = PTActiveWorkbook
    .ReportOverwriteExisting = False
End With
```

As you will see in the XDK example files, if you use VBA to change any application settings, it is always a good programming practice to restore these settings to their original values at the end of the program.

**Referencing the Model Workbook and Models**

In all likelihood, the primary reason you will automate PrecisionTree is to work with decision trees: to create them, modify them, or generate reports from them. Therefore, you might expect that you could reference a decision tree directly from the PrecisionTree root object. However, this isn’t quite how it works. There is at least one layer in between.

First, there is the PTModelWorkbook object. As its name implies, this is a reference to an Excel workbook, and it is a direct descendant of the root PrecisionTree object. It has one optional argument, a reference to the particular workbook that contains, or will contain, the decision tree. (If this argument is omitted, as it often is, the active workbook is used. Alternatively, a safer approach is to reference ThisWorkbook, a built-in Excel object that refers to the workbook that contains the code. Then you can run the code even if another workbook happens to be active.) If you want to create a new tree from scratch, the following lines are typical.

```
Dim dTree As PTDecisionTree
Set dTree = PrecisionTree.ModelWorkbook.
```

As this code indicates, a PTModelWorkbook object is obtained from the `ModelWorkbook` property. This object has a `DecisionTrees` property that returns a collection of decision trees, and you can add a member to this collection. Only the first argument of the Add method is required. It specifies the starting cell for the tree. The optional third argument specifies the name of the tree. (The second argument, True by default, indicates that the current application settings will be applied to this tree.)

Actually, as you will see in later examples in this document, you don’t need to create an explicit PTDecisionTree object (dTree in the above code). You can create it implicitly with the following With line. Then you can change properties of the tree inside the With block.

```
With PrecisionTree.ModelWorkbook.
```

Once the tree exists, you can reference it by name, but the ModelWorkbook property is still required. Here is how you could start a With block to make changes to an existing tree.

```vba
With PrecisionTree.ModelWorkbook.DecisionTrees("My Tree")
Alternatively, you can go one farther step down the hierarchy by referencing a PTModel object. This is a reference to any decision model, either a decision tree or an influence diagram. There are two ways you can reference a model, either through the Models property or through the ModelsOnWorksheet method, both members of the PTModelWorkbook object. As the names suggest, the Models method returns the collection of all models (decision trees or influence diagrams) in the workbook, whereas the ModelsOnWorksheet returns the collection of models on a specified worksheet. For example, either of the following lines starts a With block for working with an existing tree:

```vba
With PrecisionTree.ModelWorkbook.Models("My Tree")
or
With PrecisionTree.ModelWorkbook.ModelsOnWorksheet(wsTree)("My Tree")
```

This last line requires some clarification. First, it assumes that the worksheet with the tree is code-named wsTree. If the name of the worksheet, the one that appears on the worksheet tab, is Tree, you could replace wsTree with Worksheets("Tree"). Second, it might look strange to follow one argument in parentheses by another. However, the last part of the line is really equivalent to

```vba
.ModelsOnWorksheet(wsTree).Item("My Tree")
```

The Item property of a collection is always the default property and can be (and usually is) omitted—hence the two consecutive set of parentheses.

**Working with Decision Trees: Nodes and Branches**

Decision trees are by their very nature visual, and it helps to visualize their structure as you work with them in VBA code. Decision trees have nodes, followed by branches, followed by more nodes, followed by more branches, and so on. This sequence can occur in almost any order. The three types of nodes you encounter most often are decision nodes (squares), chance nodes (circles), and end nodes (triangles). The branches emanating from decision or chance nodes have values, usually monetary values (and often equal to 0), and the branches emanating from a chance node also have probabilities that sum to 1. In addition, all decision and chance nodes and branches have names—the labels that appear to the right of nodes and on the branches.

If you use VBA to build a decision tree from scratch, you must specify all of this information—node types, names, values, and probabilities—and as you will see shortly, it is quite easy to do so. However, you also have to maneuver between nodes and branches so that you can “build” the tree in the proper sequence. There are several ways to reference nodes and branches in a tree, but the following guidelines are arguably the one you will find most useful.

1. A tree has a **RootNode** property, the leftmost node in the tree. You get it “for free” as soon as you create a new tree.
2. Each node—technically a PTDecisionTreeNode object—has a **NodeType** property. The three most common node types are PTTreeDecisionNode, PTTreeChanceNode, and PTTreeEndNode.
3. Each decision and chance node has a **ChildBranches** property that references the collection of branches emanating from that node.
a. You can refer to a particular branch by its name, as in ChildBranches("Win").
b. Alternatively, you can use the Add method to add a new branch, as in
ChildBranches.Add("Win", PFirstBranch). The first argument is the name you give to the
branch, and the second argument (one of two optional arguments) indicates that this
will be the top branch out of this node.

4. Each branch—technically a PTDecisionTreeBranch object—has a **DestinationNode** property that
returns the node to the right of this branch. It also has a **SourceNode** property that returns the
node from which the branch emanates.

5. Each node and branch has a **Name** property for labeling. Note: Many trees use the *same* name,
such as Drill Decision, for multiple nodes or branches. If you plan to refer to these by name in
your code, it is best to give them distinct names, such as Drill Decision 1, Drill Decision 2, and so
on, to avoid ambiguity.

6. Each decision and chance branch has a **BranchValueCell** property, followed by a **Value** property.
This specifies the value (usually monetary) on the branch, as in BranchValueCell.Value = 1000.

7. Each chance branch also has a **BranchSelectorCell** property, followed by a **Value** property. This
specifies the probability on a chance branch, as in BranchSelectorCell.Value = 0.35.

The following macro, taken from the example file **PrecisionTree XDK – Creating Tree 1.xlsm**, illustrates
how these guidelines can be used to create a simple tree with one decision node (the root node), two
decision branches, a chance node after the first decision branch, and two chance branches from the
chance node. Again, if you visualize the tree, the code is quite straightforward. The resulting decision
tree appears in Figure 6.

```vba
Sub CreateTree()
    ' Make a new tree starting at the cell B2 in the Tree worksheet
    With PrecisionTree.ModelWorkbook.DecisionTrees
        .Add(wsTree.Range("B2"), , "Simple Gamble")
        .rootNode
            ' Single decision node
            .ChangeNodeType PTTreeDecisionNode
            .Name = "Gamble?"
            ' First decision: gamble
            With .ChildBranches.Add("Yes", PFirstBranch)
                ' Pay 100 to play
                .BranchValueCell.Value = -100
                ' Two possible outcomes: win 5000 or lose 1000
                With .DestinationNode
                    .ChangeNodeType PTTreeChanceNode
                    .Name = "Win?"
                    With .ChildBranches.Add("Yes", PFirstBranch)
                        .BranchSelectorCell.Value = 0.2
                        .BranchValueCell = 5000
                    End With
                    With .ChildBranches.Add("No", PLastBranch)
                        .BranchSelectorCell.Value = 0.8
                        .BranchValueCell = -1000
                    End With
                End With
            End With
            ' Second decision: don’t gamble, no gain, no loss
            .ChildBranches.Add "No", PLastBranch
        End With
    End With
End Sub
```
You can also use VBA to change an existing tree. As an example, the following macro, also taken from the file PrecisionTree XDK – Creating Tree 1.xlsm, can be used to change the values and probabilities in the Simple Gamble tree shown in Figure 6. Note that the code tries to reference a “Simple Gamble” tree in the With line. If there is no such tree, the macro exits with a message to this effect. Otherwise, it makes changes to the existing tree. Also, note that there is a Nodes collection object for the entire tree, so that you can get to a particular node by referencing its name. However, there is no Branches collection object for the entire tree, so to get to any branch, you have to reference the node from which the branch emanates. (There are other ways to get to a branch, but this is probably the most straightforward.)

```
Sub ChangeTree1()
    ' Quit if there is no such tree
    On Error GoTo exitPoint
    With PrecisionTree.ModelWorkbook.DecisionTrees("Simple Gamble")
        ' Change cost of playing
        ' Change probabilities of winning and losing
        .Nodes("Win?").ChildBranches("Yes").BranchSelectorCell.Value = 0.1
        .Nodes("Win?").ChildBranches("No").BranchSelectorCell.Value = 0.9
    End With
Exit Sub

exitPoint:
    MsgBox "There is no tree to change.", vbInformation
End Sub
```

The following macro, again from the file PrecisionTree XDK – Creating Tree 1.xlsm, illustrates another possible change. Now the signs of all monetary values are changed, and the OptimumPath property of the tree is changed to PTMinimumPayoff. This creates a tree that is equivalent to the original tree, but with a “minimize expected cost” criterion.

```
Sub ChangeTree2()
    ' Quit if there is no such tree
    On Error GoTo exitPoint
    With PrecisionTree.ModelWorkbook.DecisionTrees("Simple Gamble")
        ' Change so that expected cost is minimized
        .OptimumPath = PTMinimumPayoff
        ' Change sign of cost of playing
        .rootNode.ChildBranches("Yes").BranchValueCell.Value = 100
        ' Change signs of values from winning and losing
        .Nodes("Win?").ChildBranches("Yes").BranchValueCell.Value = -5000
        .Nodes("Win?").ChildBranches("No").BranchValueCell.Value = 1000
    End With
Exit Sub
```
Finally, the following macro, again from the file PrecisionTree XDK – Creating Tree 1.xlsm, illustrates how you can change the decision criterion to maximize expected utility, with an exponential utility function. In this case, you can display expected values, expected utilities, or certainty equivalents on the tree. The latter is chosen here.

```vba
Sub ChangeTree3()
    ' Quit if there is no such tree
    On Error GoTo exitPoint
    With PrecisionTree.ModelWorkbook.DecisionTrees("Simple Gamble")
        ' Change to an exponential utility function and display certainty equivalents
        With .UtilityFunction
            .Enabled = True
            .FunctionType = PTExponential
            .RValue = 500
            .Display = PTDisplayCertaintyEquivalent
        End With
    End With
End Sub

exitPoint:
    MsgBox "There is no tree to change.", vbInformation
End Sub
```

The results of these three “change” macros, all applied to the original tree, appear in Figures 7, 8, and 9.

**Figure 7 Simple Gamble with Changed Data**

![Simple Gamble with Changed Data](image1)

**Figure 8 Simple Gamble with Expected Cost Minimization**

![Simple Gamble with Expected Cost Minimization](image2)
**Figure 9 Simple Gamble with Expected Utility Maximization**

![Diagram of a simple gamble with expected utility maximization](image)

*Technical Note: BranchSelectorCell*

You might wonder at the strange name for the BranchSelectorCell property. As stated earlier, when it is followed by Value, this specifies the probability on a branch emanating from a chance node. However, it can also be used for a branch emanating from a decision (or logic) node. Then it returns the YES or NO value that determines which branch is selected—that is, followed. Even with branches out of chance nodes, its value, the probability, indicates which branch is *probabilistically* selected—hence the word “Selector” in the name.

*Technical Note: InterfaceState*

If you have automated Excel with VBA, you might have turned screen updating off, and then turned it back on, as in the following lines:

```vba
Application.ScreenUpdating = False
' Do a lot of stuff
Application.ScreenUpdating = True
```

The effect is to avoid a lot of screen flickering and to speed up the process. The PrecisionTree XDK provides a similar capability for avoiding screen flicker and improving speed with the PTInterfaceState object and **StartLongProcess** method. Before doing a complex series of operations, like building a large tree, you can use the following lines, where the argument of the Set line is text that will appear in the status bar while the tree is being created.

```vba
Dim interfaceState As PTInterfaceState
On Error GoTo exitPoint
Set interfaceState = PrecisionTree.StartLongProcess("Creating Tree...")
' Code that creates the tree (might raise an error)....
On Error GoTo 0
```

Among other things, the Set line turns off screen updating, it puts Excel into manual recalculation mode, and it turns off PrecisionTree calculations. However, after all the tree-creating code, you must restore everything to its original state with the following lines:

```vba
exitPoint:
If Not (interfaceState Is Nothing) Then interfaceState.Restore True
```

The purpose of the On Error GoTo exitPoint line is to turn on error handling. This ensures that if any errors cause the program to stop unexpectedly, the Restore method will be called. Otherwise, Excel
would stay in an “unnatural” state—for example, it would stay in manual recalculation mode. The purpose of the On Error GoTo 0 line is to turn off error handling.

The Boolean argument of the Restore method can be explained as follows. Imagine that the tree-creating code, the part now commented out, raises an error, which sends it to the exitPoint and the Restore line. If the Boolean argument is False, the error will be ignored (and cleared), so that any subsequent code will run as if the error never happened. However, if this argument is True, the Restore line will immediately throw the same error that occurred in the earlier code. This would allow you to get the original error message and have the routine stop, facilitating the debugging process.

Bayesian Revision

Beginning in PrecisionTree 6.0, you can “flip” a symmetric probability tree to implement Bayes’ rule. This ability is also available in the PrecisionTree XDK with the BayesRevision method of the rootNode object. The following macro, taken from the example file PrecisionTree XDK – Creating Tree with Bayesian Revision.xlsm, illustrates the process. The probability tree is first built with the given prior probabilities and likelihoods (which by then have been read from a Data sheet). Next, the tree is flipped with the BayesianRevision method. This uses the reordering array, which indicates that the second-stage nodes should become the first-stage node, and vice versa. Finally, the probabilities required for the final decision tree (the testProb and posterior arrays) are read from the flipped tree.

```vba
Sub CreateBayesTreeAndFlip()
    Dim i As Integer, j As Integer
    Dim reordering(1 To 2) As Long
    With PrecisionTree.ModelWorkbook.DecisionTrees
        .Add(wsBayesTree.Range("B2"), , "Tree to flip")
    With .rootNode
        .ChangeNodeType PTTreeChanceNode
        .Name = "Well type"
        For i = 1 To 3
            With .ChildBranches.Add(wellLabel(i), _
                IIf(i = 1, PTFirstBranch, PTLastBranch))
                .BranchSelectorCell.Value = prior(i)
            End With
        Next
    End With
    With .rootNode
        .ChangeNodeType PTTreeChanceNode
        .Name = "Test result"
        For j = 1 To 3
            With .ChildBranches.Add(testLabel(j), _
                IIf(j = 1, PTFirstBranch, PTLastBranch))
                .BranchSelectorCell.Value = likelihood(i, j)
            End With
        Next
    End With

    ' Flip the tree
    reordering(1) = 2
    reordering(2) = 1
    .BayesianRevision reordering

    ' Get revised probabilities from flipped tree
    For i = 1 To 3
        With .ChildBranches(i)
            testProb(i) = .BranchSelectorCell.Value
        End With
        For j = 1 To 3
            posterior(i, j) = .ChildBranches(j).BranchSelectorCell.Value
        Next
    End Sub
```
Creating Reports

Once you have a decision tree, it is easy to create Policy Suggestion or Risk Profile reports. The following macro, taken from the example file PrecisionTree XDK – Creating Reports.xlsm, illustrates some possibilities. Here are several things to note:

1. Two report application settings are changed, but the original settings are stored in variables so that they can be restored at the end of the macro. Specifically, if you plan to create multiple reports, you should set the ReportOverwriteExisting property to False; otherwise, each report will overwrite the previous report.

2. Reports are created from the NewPolicySuggestion or NewRiskProfile methods of a PTModelWorkbook object. You then work with the resulting objects, PTPolicySuggestion and PTRiskProfile, created implicitly in the With lines. A model must be specified, in this case, an existing decision tree, the “Include” properties indicate which items to include in the report, and finally, the GenerateReport method creates the report.

3. By default, the starting node for the reports is the root node of the tree. However, you can specify another starting node if you want to base the report on a particular subtree of the entire tree.

```vba
Sub CreateReports()
    Dim modelWB As PTModelWorkbook
    Dim rptPlacement As PReportPlacement
    Dim rptOverwrite As Boolean

    With PrecisionTree.ApplicationSettings
        ' Remember current settings
        rptPlacement = .ReportPlacement
        rptOverwrite = .ReportOverwriteExisting
        ' Change settings
        .ReportPlacement = PTActiveWorkbook
        ' The following must be set to False. Otherwise, each report requested below will overwrite the previous report.
        .ReportOverwriteExisting = False
    End With

    Set modelWB = PrecisionTree.ModelWorkbook

    ' Policy suggestion for entire tree
    With modelWB.NewPolicySuggestion
        Set .Model = modelWB.DecisionTrees("Oil Drilling Decisions")
        .IncludeDecisionTable = True
        .IncludeOptimalDecisionTree = True
        .GenerateReport
    End With

    ' Policy suggestion assuming decision to NOT test has been made
    With modelWB.NewPolicySuggestion
        Set .startingNode = modelWB.DecisionTrees("Oil Drilling Decisions")
            .Nodes("Drill Decision 4")
        .IncludeDecisionTable = True
        .IncludeOptimalDecisionTree = True
        .GenerateReport
    End With

    ' Risk profile for entire tree
```
With modelWB.NewRiskProfile
    Set .Model = modelWB.DecisionTrees("Oil Drilling Decisions")
    .PathsToAnalyze = PTAnalyzeOptimumPath
    .IncludeCumulativeChart = False
    .IncludeProbabilityChart = True
    .IncludeStatisticalSummary = True
    .GenerateReport
End With
' Risk profile assuming decision to not test has been made
With modelWB.NewRiskProfile
    Set .startingNode = modelWB.DecisionTrees("Oil Drilling Decisions")
        .Nodes("Drill Decision 4")
    .PathsToAnalyze = PTAnalyzeOptimumPath
    .IncludeCumulativeChart = False
    .IncludeProbabilityChart = True
    .IncludeStatisticalSummary = True
    .GenerateReport
End With
With PrecisionTree.ApplicationSettings
    ' Restore original settings
    .ReportPlacement = rptPlacement
    .ReportOverwriteExisting = rptOverwrite
End With
End Sub

**Technical Note: Referring to Nodes by Cell Address**

There was a suggestion earlier in this guide that you should provide nodes with distinct names so that you can refer to them unambiguously in your code. For example, this enables the reference to “Drill Decision 4” in the above code, which wouldn’t be possible if all four decision nodes had the same “Drill Decision” name. However, there is another way to do it. You can refer to a node by its cell address. If you look closely at a PrecisionTree decision tree (with Excel gridlines turned on), you will notice that every node “straddles” two cells, one above and one below. When referring to the node, you can refer to *either* of these cells, using the `ItemFromCell` property. For example, because the Drill Decision 4 node straddles cells D50 and D51 in the example file, you could replace `.Nodes("Drill Decision 4")` with either

```vba
.Nodes.ItemFromCell(Range("D50"))
```

or

```vba
.Nodes.ItemFromCell(Range("D51"))
```

Of course, if either of these cells is range-named (probably for this specific purpose), you could instead reference the range name.

**Performing Sensitivity Analyses**

The VBA code required to perform one or more sensitivity analyses is also fairly straightforward. This is illustrated in the following macro, taken from the example file *PrecisionTree XDK – Sensitivity Analyses.xlsm*. In addition to the guidelines for reports (not repeated here), the following guidelines apply:

1. A new sensitivity analysis—technically a PTSensitivityAnalysis object—is created implicitly in the With lines with the `NewSensitivityAnalysis` method of a PTModelWorkbook object.
2. The `AnalysisType` property must be set to PTOneWayAnalysis or PTTwoWayAnalysis.
3. The starting node of the `Output` property should be set, either to the root node or some other node.
4. You then add one or more input cells and specify how you want them to vary. Also, for a two-way analysis, two inputs should be specified as the X-axis and Y-axis variables.

5. The “Include” properties depend on the type of analysis. For example, tornado and spider graphs are not applicable for a one-way analysis with a single input or a two-way analysis.

Sub PerformSensitivityAnalyses()
    Dim modelWB As PTModelWorkbook
    Dim rptPlacement As PTReportPlacement
    Dim rptOverwrite As Boolean

    With PrecisionTree.ApplicationSettings
        ' Remember current settings
        rptPlacement = .ReportPlacement
        rptOverwrite = .ReportOverwriteExisting
    ' Change settings
        .ReportPlacement = PTActiveWorkbook
        ' The following must be set to False. Otherwise, each report
        ' requested below will overwrite the previous report.
        .ReportOverwriteExisting = False
    End With

    Set modelWB = PrecisionTree.ModelWorkbook

    ' One-way sensitivity analysis from root node on a single input
    With modelWB.NewSensitivityAnalysis
        .analysisType = PTOneWayAnalysis
        .IncludeStrategyRegion = True
        .IncludeSensitivityGraph = True
        .GraphsDisplayPercentageChange = False
        Set .Output.startingNode = modelWB.DecisionTrees("Oil Drilling Decisions").rootNode
        With .Inputs.Add
            Set .VaryCell = Range("Cost_of_test")
            .VariationMethod = PTActualMinMaxValues
            .BaseValue = Range("Cost_of_test").Value
            .Minimum = .BaseValue - 10000
            .Maximum = .BaseValue + 50000
            .Steps = 7
        End With
        .GenerateReport
    End With

    ' One-way sensitivity analysis from an interior node on a single input
    With modelWB.NewSensitivityAnalysis
        .analysisType = PTOneWayAnalysis
        .IncludeStrategyRegion = True
        .IncludeSensitivityGraph = True
        .GraphsDisplayPercentageChange = False
        Set .Output.startingNode = modelWB.DecisionTrees("Oil Drilling Decisions").Nodes("Drill Decision 1")
        With .Inputs.Add
            Set .VaryCell = Range("Cost_of_drilling")
            .VariationMethod = PTActualMinMaxValues
            .BaseValue = Range("Cost_of_drilling").Value
            .Minimum = .BaseValue - 100000
            .Maximum = .BaseValue + 100000
            .Steps = 7
        End With
        .GenerateReport
    End With

    ' One-way sensitivity analysis from root node on several inputs
    With modelWB.NewSensitivityAnalysis
        .analysisType = PTOneWayAnalysis
        .IncludeStrategyRegion = False
        .IncludeSensitivityGraph = False
        .IncludeTornadoGraph = True
        .IncludeSpiderGraph = True
        .GraphsDisplayPercentageChange = False
        Set .Output.startingNode = modelWB.DecisionTrees("Oil Drilling Decisions").rootNode
    End With
End Sub
With .Inputs.Add
    Set .VaryCell = Range("Cost_of_test")
    .VariationMethod = PTActualMinMaxValues
    .BaseValue = Range("Cost_of_test").Value
    .Minimum = .BaseValue - 10000
    .Maximum = .BaseValue + 50000
    .Steps = 7
End With
With .Inputs.Add
    Set .VaryCell = Range("Cost_of_drilling")
    .VariationMethod = PTActualMinMaxValues
    .BaseValue = Range("Cost_of_drilling").Value
    .Minimum = .BaseValue - 500000
    .Maximum = .BaseValue + 100000
    .Steps = 7
End With
With .Inputs.Add
    Set .VaryCell = Range("Small_well_value")
    .VariationMethod = PTPercentageChangeFromBase
    .BaseValue = Range("Small_well_value").Value
    .Minimum = -20
    .Maximum = 40
    .Steps = 7
End With
With .Inputs.Add
    Set .VaryCell = Range("Large_well_value")
    .VariationMethod = PTPercentageChangeFromBase
    .BaseValue = Range("Large_well_value").Value
    .Minimum = -20
    .Maximum = 40
    .Steps = 7
End With
GenerateReport
End With

' Two-way sensitivity analysis from root node
With modelWB.NewSensitivityAnalysis
    .analysisType = PTTwoWayAnalysis
    .IncludeStrategyRegion = True
    .IncludeSensitivityGraph = True
    .GraphsDisplayPercentageChange = False
    Set .Output.startingNode = modelWB.DecisionTrees("Oil Drilling Decisions").rootNode
With .Inputs.Add
    ' Each input must be designated for the X-axis or the Y-axis
    .TwoWayAnalysis = PTTwoWaySensitivityXAxis
    Set .VaryCell = Range("Small_well_value")
    .VariationMethod = PTPercentageChangeFromBase
    .BaseValue = Range("Small_well_value").Value
    .Minimum = -20
    .Maximum = 40
    .Steps = 7
End With
With .Inputs.Add
    .TwoWayAnalysis = PTTwoWaySensitivityYAxis
    Set .VaryCell = Range("Large_well_value")
    .VariationMethod = PTPercentageChangeFromBase
    .BaseValue = Range("Large_well_value").Value
    .Minimum = -20
    .Maximum = 40
    .Steps = 7
End With
GenerateReport
End With

With PrecisionTree.ApplicationSettings
    ' Restore original settings
    .ReportPlacement = rptPlacement
    .ReportOverwriteExisting = rptOverwrite
End With
Some General VBA Tips

This guide concludes with a few VBA tips that you should know regardless of whether you are automating PrecisionTree or Excel.

File Format (xlsm)

If you save a workbook that includes VBA code, you must save it as a macro-enabled (.xlsm) file. This is true for any Excel file that contains VBA code; it is not specific to PrecisionTree files. Then if you open this .xlsm file later on, you will be warned that it contains macros. Make sure you elect to enable the macros; otherwise the VBA macros won’t work.

Running a Macro

If you develop a VBA program for nontechnical users, you probably won’t want them to see your code, either for proprietary reasons or because it would be too intimidating. However, it is easy to create a simple user interface for running the program. To do so, activate a worksheet, insert a shape such as a rectangle, right-click it, select Assign Macro, and select your macro. You can also insert descriptive text, such as Run Program, in the shape. From then on, a user can simply click the shape to run the program.

ThisWorkbook

You can always reference a workbook by name, as in Workbooks(“My Example File.xlsm”). Alternatively, you can refer to the active workbook with the built-in object ActiveWorkbook. However, an even safer reference is to ThisWorkbook, which always references the workbook containing the VBA code. (It is safer than ActiveWorkbook because the workbook containing the code might not be active when you run your program.)

Worksheet Code Names

The code in the example files sometimes references worksheets by their “code” names. A worksheet actually has two names, the name you see on its tab, and a code name, which can be set only in the Visual Basic Editor. This is illustrated in Figure 10 (see the highlighted line in the Properties section). In this example, the notation wsTree (Tree) in the Project section indicates that Tree is the name on the tab and wsTree is the code name. (Any names can be used for code names, but a common practice is to use the prefix ws.) One reason for using code names in VBA programs is that you don’t need to create a Worksheet object variable; you can simply write wsTree.Range(“A1”), for example. A second reason is that if your code refers to the worksheet by name, as in Worksheets(“Tree”), and someone changes the name on the worksheet tab, an innocent enough change to make, this will break your program. However, because code names can be changed only through the Visual Basic Editor, it is much less likely that anyone will change them.
In the XDK example files, you will see a lot of `With` blocks, sections that start with the keyword `With` and end with the keyword `End With`. These are used by all good VBA programmers, so you should get used to them. They allow you to avoid a lot of repetitive code, as illustrated by the following block of code (where `dTree` has been declared as a `PTDecisionTree` object variable). Inside the `With dTree` block, anything that starts with a period really has `dTree` to the left of it. For example, `.Name` is equivalent to `dTree.Name`. As you can see, these `With` blocks can be nested to avoid even more repetitive code. For example, `.ChildBranches` in the third-to-last line is equivalent to `dTree.rootNode.ChildBranches`. It is a very good idea to indent these blocks appropriately, as has been done here, for easier reading.

```
With dTree
  With .rootNode
    .ChangeNodeType PTTreeDecisionNode
    .Name = "Gamble?"
    With .ChildBranches.Add("Yes", PTFirstBranch)
      .BranchValueCell.Value = -100
      With .DestinationNode
        .ChangeNodeType PTTreeChanceNode
        .Name = "Win?"
        With .ChildBranches.Add("Yes", PTFirstBranch)
          .BranchSelectorCell.Value = 0.2
          .BranchValueCell = 5000
        End With
        With .ChildBranches.Add("No", PTLastBranch)
          .BranchSelectorCell.Value = 0.8
          .BranchValueCell = -1000
        End With
      End With
    End With
  End With
End With
```

**Figure 10 Code Name for Worksheet**

![Image](image.png)
Adding to a Collection

Virtually all collection objects have an Add method for adding an item to the collection. For example, the Worksheets collection has an Add method for adding a worksheet to a workbook. The following code is typical. It adds a new worksheet after the last worksheet and gives it a couple properties. The Add line is compact in that it not only adds the worksheet (and makes it the active worksheet), but it returns a reference to the new worksheet so that the ws object variable can be “set.”

```vba
Dim ws As Worksheet
Set ws = Worksheets.Add(after:=Worksheets(Worksheets.Count))
With ws
    .Name = "Revenues"
    .Range("A1").Value = "Revenues for our new product"
End With
```

If the ws object variable is not needed later in the program, this code can be compacted even further, as follows. Now there is no ws object variable. The With line creates an implicit reference to the new worksheet, so that its properties can be referenced inside the With block.

```vba
With Worksheets.Add(after:=Worksheets(Worksheets.Count))
    .Name = "Revenues"
    .Range("A1").Value = "Revenues for our new product"
End With
```

A similar construction can be used with PrecisionTree code. The following is typical. The first line uses the Add method to create a new branch from the root node, and everything inside the outer With block is relative to this new branch. Then after the second With block references the new branch’s destination node, the next two With blocks use the Add method to add and then set properties of two more new branches.

```vba
With dTree.RootNode.ChildBranches.Add("Yes", PTFirstBranch)
    .BranchValueCell.Value = -100
    .ChangeNodeType PTTreeChanceNode
    With .DestinationNode
        With .ChildBranches.Add("Yes", PTFirstBranch)
            .BranchSelectorCell.Value = 0.2
            .BranchValueCell = 5000
        End With
        With .ChildBranches.Add("No", PTLastBranch)
            .BranchSelectorCell.Value = 0.8
            .BranchValueCell = -1000
        End With
    End With
End With
End With
End With
```

Intellisense

The reference to the PrecisionTree library mentioned earlier is not only required to make your PrecisionTree code work properly, but it provides you with an invaluable programming aid: Intellisense. As an example, consider the following line (where dTree is again assumed be declared as a PTDecisionTree object variable):

```vba
dTree.UtilityFunction.FunctionType = PTExponential
```

Could you remember all of this? Fortunately, you don’t have to. As soon as you type dTree. (including the period), you will see a list you can choose from. Then when you choose UtilityFunction from the list and type a period, you will see another list to choose from. And so on. To cap it off, when you have entered everything to the left of the equals sign and then type the equals sign, you will see a list of the
appropriate PrecisionTree constants for the FunctionType property that you can choose from. If you are already an Excel programmer, you undoubtedly appreciate the power of Intellisense, and you will be happy to know that it extends to VBA for PrecisionTree. If you are not yet familiar with Intellisense, you are in for a big treat!

**Built-In Constants**

The line `dTree.UtilityFunction.FunctionType = PTExponential` contains one of many built-in PrecisionTree constants, in this case, PTExponential. All of these constants begin with PT, and they all have integer values, such as 3 or 445. However, because no person could ever remember these values, they are instead given meaningful names that programmers can use in their VBA code. You can find the names in the Object Browser (discussed next). For example, Figure 11 lists the constants (on the right) for the three possible settings of the FunctionType property. Together, this set of three constants is called an enumeration. (Strictly speaking, the FunctionType property returns the PTUtilityFunctionType enumeration. This is only one of several enumerations available in PrecisionTree, indicated by the double-yellow icons in the figure. ³)

**Figure 11 PTUtilityFunctionType Enumeration**

![Object Browser](image)

**Object Browser**

A great source of information is the VBA Object Browser. To see this, make sure a file with a reference to the PrecisionTree library is selected in Project pane. Then click the circled button in Figure 12, and select PTreeOL6 from the libraries dropdown list. This Object Browser shows all of the PrecisionTree objects on the left and all of their properties and methods on the right. Also, when you select an item on the right,

---

³ Excel also has many enumerations. Its built-in constants all begin with xl, as in xlDown, xlToLeft, and so on.
you get some limited help, such as the list of a function’s arguments, at the bottom of the window. Remember that such help is only a few mouse clicks away as you are doing your programming. It often provides just the detail you need to get you past a programming hurdle.

**Figure 12 Visual Basic Object Browser**

![Image of Visual Basic Object Browser]

**Appendix**

The document to this point contains the information you need to get started with VBA for PrecisionTree. This appendix contains extra information, some of which is rather technical, that might come in handy.

**Updating PrecisionTree 5.x Automation Code to Run in PrecisionTree 6.x**

If you have automation code written for PrecisionTree 5.x, that code should continue to work with PrecisionTree 6.x, *provided that you set the references as described earlier*. Likewise, most code written for PrecisionTree 6.x will work in 5.x if you make the appropriate reference changes and you don’t use any of the new properties and methods that were added in version 6.x.

The required references for PrecisionTree 6.x are somewhat different from PrecisionTree 5.x. In the earlier version, you made only a single reference to the PrecisionTree 5.x for Excel Object Library. If you are converting automation code designed for use with PrecisionTree 5.x to PrecisionTree 6.x, you must remove this old reference and make the two references described earlier.

Also, to automate PrecisionTree version 5.7 in 64-bit Excel, you were required to include a block of code to obtain the root object and to access various constant values used in the automation library. This code block is no longer needed and should be removed from your code.
Demand-Loading PrecisionTree

You usually want PrecisionTree to load automatically when you open your files containing PrecisionTree macros. By adding the reference to PtreeXLA in the VBA references as described earlier, this will occur. However, there are some instances where this is not the behavior you want. For example, you might want to create an application that uses PrecisionTree only part of the time. Then requiring PrecisionTree to always be loaded is not appropriate. In such cases, the loading and shutdown of PrecisionTree itself is your code’s responsibility.

If you remove the PtreeXLA reference, which is really a reference to the Ptree.xla add-in file, you can still automate PrecisionTree, provided that you load PrecisionTree yourself, but you need to do some extra work to obtain the root PrecisionTree reference object and to use any PrecisionTree constants defined in the Ptree.xla file.

You can get the root PrecisionTree object in a “late-bound” fashion—that is, without an early-bound reference to PtreeXLA—by using Excel’s Application.Run command as follows:

```vba
Dim PrecisionTree As PtreeOL6.PrecisionTree
Set PrecisionTree = Application.Run("Ptree.xla!PrecisionTree")
```

This code assumes that PrecisionTree is already loaded, and it calls the public property PrecisionTree in a late-bound fashion to access the root automation object. If you store PrecisionTree as a global (non-local) variable, we recommend that you set it to Nothing when you are finished using it for automation. Actually, you could call this variable anything, but by using the variable name PrecisionTree, your code will only be minimally changed.

Another purpose the early-bound reference to Ptree.xla serves is to provide all the automation constants (PTErrorValue, PTNegativeInfinity, and so on) that you might need for your code. For example, in the following line of code, the constant PTErrorValue is provided by the reference to Ptree.xla.

```vba
If PrecisionTree.ModelWorkbook.DecisionTrees("MyTree").CalculateOutputDistribution().StdDev = _
    PTErrorValue Then MsgBox "Error Value."
```

If you choose to demand load PrecisionTree, however, and you do not have a reference to Ptree.xla, these constants won’t be available directly. Instead you will need to access them from the PrecisionTree.Constants object, as follows:

```vba
If PrecisionTree.ModelWorkbook.DecisionTrees("MyTree").CalculateOutputDistribution().StdDev = _
    PrecisionTree.Constants.PTErrorValue Then MsgBox "Error Value."
```

Automating PrecisionTree from .NET

Although the PrecisionTree automation library is primarily designed for use from within Excel’s VBA language, and all of the code examples of this documentation, excepting this section, are written for VBA, it is also possible to use the .NET environment to control PrecisionTree. To automate PrecisionTree from .NET, you must first add a reference in the Visual Studio "Add Reference" dialog. In the COM tab of that dialog, you should select "Palisade PrecisionTree 6.x for Excel Object Library".

Because you can’t set a reference to Ptree.xla in .NET, the same issues described above for late-bound loading of PrecisionTree apply for .NET programmers. Specifically, you must access the root PrecisionTree object manually, you must access PrecisionTree constants indirectly, and PrecisionTree won’t automatically be loaded for you.
Assuming that PrecisionTree is loaded, you obtain the root PrecisionTree object through a COM call to Excel and then store it in an object variable. You can choose any name for this variable, but the recommended name is PrecisionTree. With this name, automation code written for .NET will be very similar to the VBA code examples illustrated in this document.

As with the demand-loaded PrecisionTree described earlier, all PrecisionTree constants must be accessed via the PrecisionTree .Constants object.

Here are examples of C# and VB .NET methods for a simple PrecisionTree automation. Assuming it is passed a reference to Excel that already has PrecisionTree loaded, it sets up the root PrecisionTree object, and then it creates a small tree. These methods could be part of .NET class libraries, and those libraries could be accessed from Excel using COM/.NET Interoperability if desired.

Note: To automate Excel, an appropriate reference and "using"/"Imports" lines need to be added. You can consult relevant Microsoft documentation for details.

C# Code

```csharp
//This code assumes "using PtreeOL6;" line has been added, as well as an appropriate "using" line to access Excel automation interface.
public void CreateTree(Application ExcelApplication) {
    PrecisionTree PrecisionTree;
    //Unfortunately C# has no support for missing arguments, thus this call to Excel’s Application.Run (which takes up to 30 optional arguments after the macro name) method is very ugly.
    //"Missing.Value" can be used after "using System.Reflection;" has been added.
    PrecisionTree = (PrecisionTree)ExcelApplication.Run("Ptree.xla!PrecisionTree", Missing.Value, 
    
    "SimpleTree")
}
```

VB .NET Code

```vbnet
' This code assumes that "Imports PtreeOL6" line has been added, as well as an appropriate 
' Imports line to access Excel automation interface.
Public Sub CreateTree(ByVal ExcelApplication As Application)
    Dim PrecisionTree As PrecisionTree
    
    PrecisionTree = ExcelApplication.Run("Ptree.xla!PrecisionTree ")
    "SimpleTree")
End Sub
```