Numeric Windows Components

CompuKalc Ltd, 2002

email: compukalc.com@ntlworld.com

Keywords:

ActiveX Components, Microsoft Office, Microsoft Windows, Internet, Numerical Computation, Numerical Optimization.

1 Introduction

In recent years there has been a trend towards the use of mathematical software within the Microsoft Windows environment. It is because of this that suppliers of mathematical software need to carefully consider the manner in which their product is delivered. Central to this is the issue of the software's user-interface.

One approach is to provide Dynamic Link Libraries (DLLs) which allow Mathematical software to be used from within Excel, Visual Basic, Visual C++, Visual Fortran etc. However, to call a DLL routine directly from Visual Basic requires detailed knowledge of both the routine's arguments and also the manner in which they are passed to the Visual Basic calling program. It is therefore essential that users have access to all the relevant documentation. This approach also has the following disadvantages.

• Currently there are certain restrictions on the use of DLL routines, for instance they cannot be incorporated into an HTML Web page.

• DLLs are not in the spirit of Microsoft's object-based approach to programming and do not make use of that technology.

• They must be called using low level program statements and cannot be accessed interactively or visually.

By using an Excel Add-In to provide a higher level user-interface to the underlying DLL it is possible to alleviate some of the difficulties previously mentioned. However, it should be mentioned that:

• Not all versions of Excel are compatible.

• There remains the issue of how potential users are to access routines from Visual Basic, Delphi, PowerPoint, etc.

• The underlying framework of the Excel user-interface cannot be changed (since it was created by Microsoft) and can appear rather tedious for routines with large argument lists, etc.

So, what is the natural interface to use within Microsoft Windows? Ideally what is needed is an easy to use interface that would allow all routines to be called from every Microsoft product. In fact such an interface does already exist: the Component Object Model (COM). It is used by Microsoft, Inprise, Digital Equipment Corporation, and many other companies.

Microsoft has also created the COM-based technologies of ActiveX and OLE to allow Microsoft users the ability to interact with their environment. All the mathematical software described here could have been deployed using custom (user-defined) COM interfaces. However these non-standard COM interfaces would then require separate documentation and would not automatically integrate into Microsoft products such as Visual Basic, Visual C++, etc. To avoid these problems only the standard Automation interface IDispatch (see Section 2) will be considered here. The IDispatch COM interface allows ActiveX components to be easily used from languages such as Visual Basic, VBScript, and Inprise Delphi. It also permits easy incorporation of mathematical software into Excel, Word, PowerPoint, Access, and HTML Web pages. ActiveX components can also be used from Visual C++, Visual J++, and Visual Fortran.

Some of the advantages of ActiveX components are:

• They can be used by the complete range of Microsoft products and also by other Windows software such as Inprise Delphi.

• They support drag and drop technology and so can easily be incorporated into an application.

• The properties, methods and events of a given ActiveX component can be viewed using the Microsoft (Inprise) Object Browser.

• Their object-based C++ technology can be used to provide simple user-interfaces to otherwise complicated routines.

The last point refers to the complete range of C++ class/object based technology. This includes optional arguments with default values, data/information hiding within the object, object initialization via constructors, and the properties, methods and events supported by an object.

Although ActiveX technology is currently used extensively for providing visual user-interfaces there are few examples of its use to construct mathematical or numeric components [1][2].

This paper provides detailed information on how computational ActiveX components can be incorporated into both Microsoft and Inprise products. Illustrative figures and examples of working source code are also supplied.

2 COM and ActiveX Controls

This section includes a brief outline of the basic principles of COM and how ActiveX controls are accessed from Visual Basic and Visual C++ using the IDispatch interface (also called dispinterface for short). There is insufficient space to fully explain everything mentioned in this section, but comprehensive information can be obtained from the available literature on COM and ActiveX [3-7].

ActiveX controls are DLL servers that need to be registered in the Windows Registry before they can be dynamically linked by a client. Every registered ActiveX control has a unique class identifier (CLSID) which allows a client to load it from the DLL in which it resides and create an instance of the component.

ActiveX controls are COM objects that usually have a visual user-interface and also support a variety of interfaces including those that allow Automation and events.

Automation allows an ActiveX control's properties and methods to be accessed programmatically from a language such as Visual Basic or C++, and is implemented using the IDispatch COM interface. Event-handling for events such as Single (Double) Click is implemented using COM interfaces such as IConnectionPoint and IConnectionPointContainer.

Since the main purpose of the components described here is to perform numeric calculations they only need a restricted visual user-interface and will therefore be called primitive ActiveX components. In fact, a control that maintains an on-screen window must manage messages for the window and is therefore slower than a windowless control. These primitive controls are ideal for use as numeric engine components since their limited visual user-interface will not interfere with the user-interface of the application into which they are embedded. Mathematical applications with sophisticated user-interfaces can therefore readily be constructed through the incorporation of primitive ActiveX components.

Example User-Interfaces

This section of the paper considers the user-interfaces provided by the following three example ActiveX controls:

- A Fourier Transform Control
- A Financial Derivative Pricing Control
- A Numerical Optimization Control

Each of these examples was chosen to emphasize particular features in the user-interface design and control creation. They are not to be taken as representative samples of a commerical product, but are merely given here to illustrate the type of user-interface achievable using ActiveX. The controls are discussed in terms of their interactive user-interface and language user-interface. The interactive user-interface consists of user-events, the visual design-time user-interface, and the visual run-time user-interface. The language user-interface is the Automation IDispatch interface (discussed in Section 2) and is illustrated for convenience using Visual Basic code.

The FFT control was designed to be interactive and is very suitable for small calculator type problems which might occur when giving an introductory undergraduate course on FFTs. It does however also have a language user-interface that allows it to perform FFTs on long data sequences. It is an example of an aggregated control (the control incorporates other controls such as buttons and textboxes) and was created using Visual Basic. No numerical computations were performed using Visual Basic within the control; they were carried out in Visual C++.

The financial derivative pricing control illustrates a control that gives similar importance to both its language and interactive user-interfaces. Its properties can be set interactively at design-time and have associated events, properties and methods. The control was created using Visual C++ and all the numerical computations are performed (in Visual C++) within the control.

The numerical optimization control illustrates how ActiveX can improve the user-interface to complex numerical software that requires user-defined function arguments. The control was designed primarily for use within Visual Basic, HTML Web pages (using VBScript), Excel and (Inprise) Delphi.

It is acknowledged that commerical versions of the example software would require,

• Comprehensive documentation, both printed and as Help file information.

• Sophisticated interactive design-time and language user-interfaces.

• To take account of efficiency considerations. For example the fast Fourier transform control described here was created using Visual Basic and inputs its data as a string and then accesses an external C++ DLL routine for numeric computations. A commercial version of this control would probably be created using Visual C++, input its data as an array argument and perform all the numerical calculations in C++ without reference to an external DLL.

The examples are provided merely as a guide to show what is possible using ActiveX and should not be regarded as a definitive statement on what constitutes a good user-interface.

3.1 The Fourier Transform Control

This example illustrates the use of an ActiveX control which performs a fast Fourier Transform (FFT) on a given input sequence [8]. Although it was primarily designed to be used as an educational interactive calculator it can also (via its language user-interface) be used for larger scale problems. The control was created using Visual Basic and calls a C Library routine, contained within a DLL, to perform the numeric computations. It is contained in the file `FFT.ocx` and has an associated Help file which explains how to use the control.

This control is an example of an aggregated component and was constructed using the following standard Microsoft ActiveX components: a Visual Basic form, textboxes, and command buttons. Since this control contains several other (visible) ActiveX controls it is not a (visually) primitive ActiveX component.

Note that when the command button Calculate is clicked, data is input from the textbox `Input_sequence` processed and then output to the textbox `Output_results`. If interactive use is not convenient then the control can also process data programmatically with its DataX property.

The visual appearance of the control within a Word document is shown in Figure 1. The sentence "This is an example of using ActiveX software ..." is Word document text and does not form part of the control.

3.1.1 Interactive User-Interface

The interactive use of the control is illustrated in Figure 1 and Figure 2. A sequence of real numbers is input and the fast Fourier transform is returned by clicking the Calculate button.

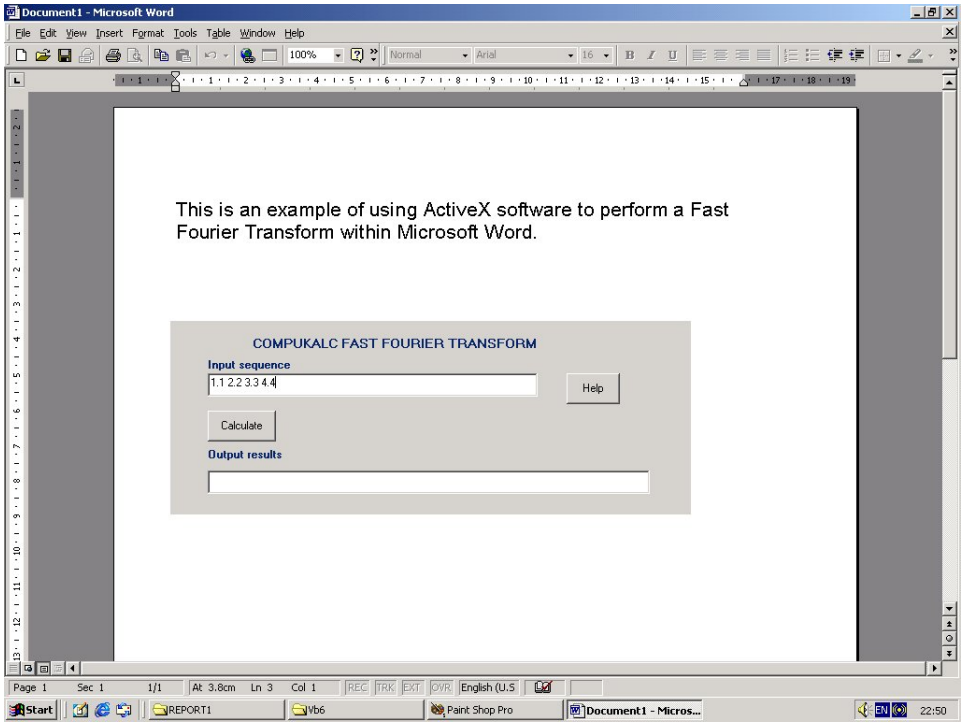

Figure 1: Microsoft word before the fast Fourier transform has been calculated.

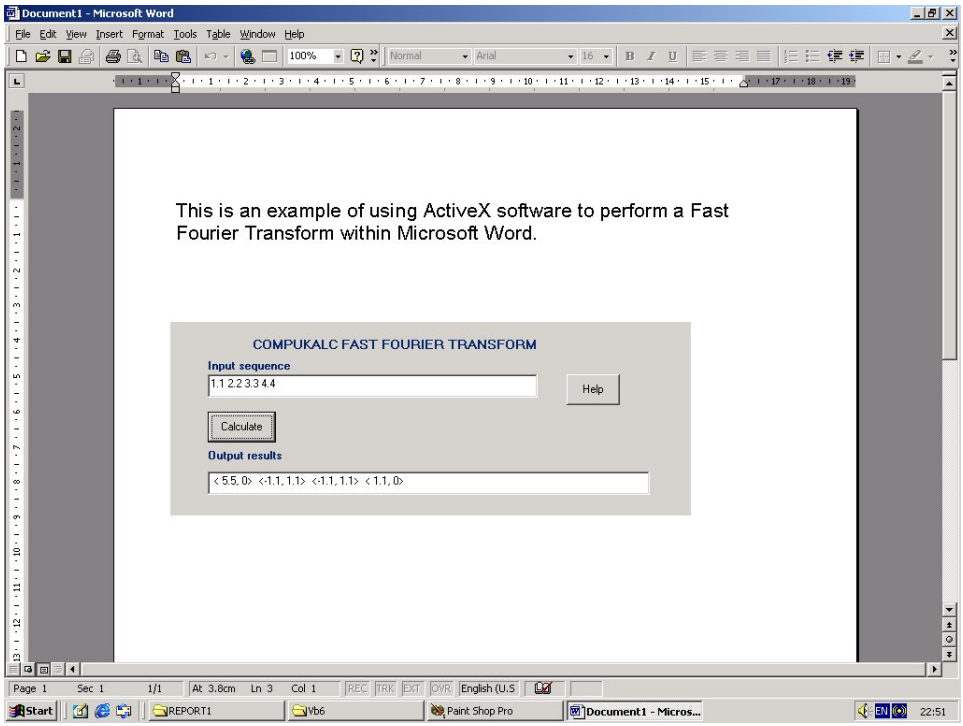

Figure 2: Microsoft word after the fast Fourier transform has been calculated.

3.1.2 Language User-Interface

The following Visual Basic code excerpt illustrates two ways in which the control can be accessed from Visual Basic. In the first method, `CreateObject` is used to access the registry and to create a new instance of the control called `myfft`. The control's `DataX` property is then used to compute a fast Fourier transform. In the second method, the control instance `FOURIER1`, created by using the Visual Basic Form Controls toolbar, is used to perform the computations.

```
Private Sub Form_Click()
   Dim myfft As Object
   Dim mystr As String
   Dim mystr1 As String

  ' Method 1.  Create a new instance of the control
    Set myfft = CreateObject("FFT.FOURIER")

    ' Perform prior calculations and programmatically construct the data string
    ' for the FFT.
    ' Note : The data string can be up to 2^32 characters long.

    mystr = "1.0 2.0 3.0 4.0 5.0" ' Example of hard coded data

    ' Use the DataX method to compute the FFT
    myfft.DataX = mystr    ' Input the data sequence
    mystr1 = myfft.DataX   ' Return the computed results
    ' Perform subsequent calculations

    ' Method 2. Use FOURIER1, an existing instance of the control
    ' created by using the Visual Basic Form Controls toolbar to interactively place
```

```
    ' the ActiveX control onto a Visual Basic form.

    FOURIER1.DataX = mystr
    mystr1 = FOURIER1.DataX
    ' Perform subsequent calculations

End Sub
```

3.2 The Financial Derivative Control

This control was created using Visual C++. It calculates the value of a financial derivative (option) by solving the Black-Scholes Partial Differential Equation [9-12]. The control contains all the necessary numeric C++ code. The control is contained in the file `DERIV.ocx`, and its instance in this Visual Basic example is called `DERIV1`.

3.2.1 Interactive User-Interface

The interactive user-interface includes Property values that can be set using the Microsoft Properties Window and also Events.

Here the ActiveX control uses an event to initiate computation at run-time. No calculations are performed until the control has been clicked by the mouse, as shown in Figure 3.

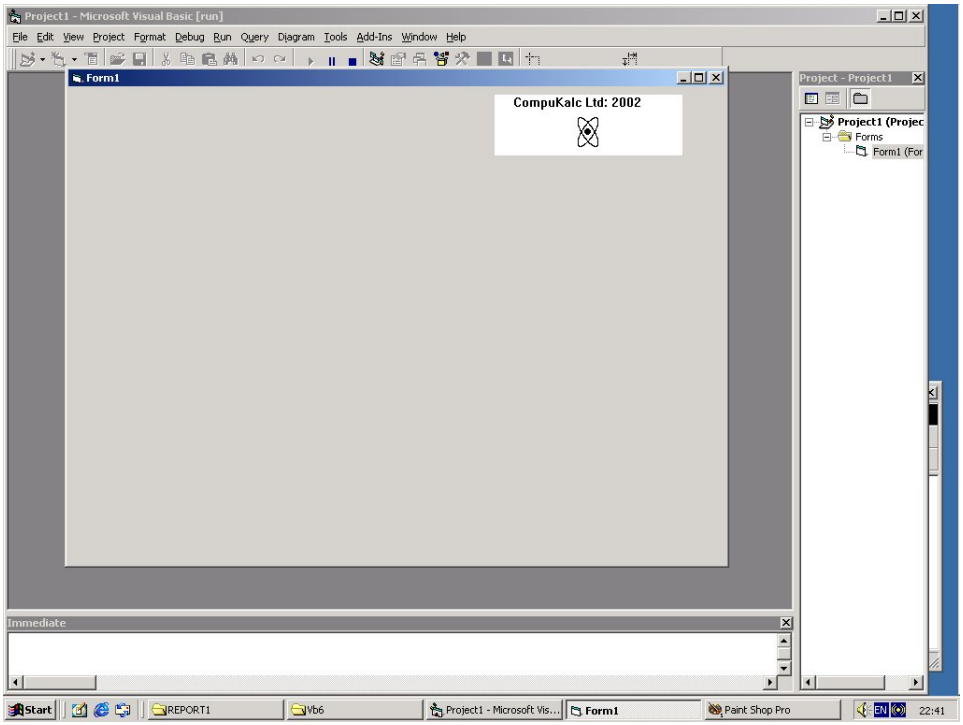

Figure 3: The user form and control before computations are performed.

Once the control has been clicked, the subroutine `DERIV1_Click()` is invoked and computations are performed; see Figure 4. The source code within `DERIV1_Click()` is given below in Section 3.2.1.

3.2.1 Language User-Interface

When the control DERIV1 is placed on the user's form, Visual Basic will automatically provide the following template code:

```
Private Sub DERIV1_Click()


End Sub
```

This subroutine is run whenever the control DERIV1 is clicked. The subroutine contains the following code:

```
Private Sub DERIV1_Click()

 Dim greeks(3) As Double
 Dim S0 As Double
 Dim r As Double
 Dim q As Double
 Dim sigma As Double
 Dim T As Double
 Dim x As Double
 Dim maturity As Double
 Dim i As Long

   x = 8#
    S0 = 10#
    r = 0.1
    sigma = 0.3
    q = 0.06

    Font.Bold = True
    Font.Size = 14
    Print " "
    Print " "
    Print "AMERICAN PUT OPTIONS "
    Print " "
    Print " Time        Option Value          Delta                Gamma           Theta"
    Print "(Years)"

    DERIV1.putcall = 1          ' A put option
    DERIV1.curval = S0          ' The current asset value
    DERIV1.strike = x           ' The strike price
    DERIV1.dividends = q        ' The continuous dividend yield
    DERIV1.method = 0           ' Use the standard lattice
    DERIV1.numsteps = 10        ' The number of time steps
    DERIV1.intrate = r          ' The risk free interest rate
    DERIV1.extype = 1           ' An american option
    DERIV1.sigma = sigma        ' The volatility

    ' Construct a table of option values and greeks for different maturities
    For i = 1 To 3
      T = i * 0.25
      DERIV1.maturity = T       ' The maturity, in years
      DERIV1.Calculate          ' Do the calculations
      opt_val = DERIV1.optval   ' Get the value of the option
```

```
      DERIV1.greeks greeks(0)  ' Get the calculated hedge statistics (greeks)

      ' Now output the results in tabular format
        Print " "; Format(T, "#0.00"), Format(opt_val, "#0.0000"), _
        Format(greeks(0), "#0.0000"), Format(greeks(1), "#0.0000"), _
        Format(greeks(2), "#0.0000")

    Next i

    Print " "
    Print "AMERICAN PUT OPTIONS  (USING CONTROL VARIATE)"
    Print " "
    DERIV1.extype = 2            ' An option, calculated using the control variate method

    Print " Time        Option Value          Delta                Gamma           Theta"
    Print "(Years)"

    ' Construct a table of options values and greeks for different maturities
    For i = 1 To 3
      T = i * 0.25
      DERIV1.maturity = T         ' The maturity in years
      DERIV1.Calculate            ' Do the calculation
      opt_val = DERIV1.optval     ' Get the value of the option
      DERIV1.greeks greeks(0)     ' Get the calculated hedge statistics (greeks)

      ' Now output the results in tabular format
      Print " "; Format(T, "#0.00"), Format(opt_val, "#0.0000"), _
      Format(greeks(0), "#0.0000"), Format(greeks(1), "#0.0000"), _
      Format(greeks(2), "#0.0000")
    Next i

End Sub
```

The code illustrates that the properties `DERIV1.putcall`, `DERIV1.curval`, `DERIV1.sigma`, etc are used to set up the values for the problem. The method `DERIV1.calculate` then performs the required calculations, and option values and greeks are returned via the property `DERIV1.optval` and method `DERIV1.greeks`, respectively. From the output in Figure 4, using `DERIV1.extype` = 1 and `DERIV1.extype` = 2 results in slightly different option values and hedge statistics (greeks), because here the partial differential equation is approximated using a lattice with only ten time steps. The most accurate values are expected to be those calculated using the Control Variate method. This method uses the analytic value of the corresponding european option to adjust the answers returned by the lattice. However, as the number of time steps is increased while maintaining a fixed time integration interval, the results from both methods converge and to the same answer.

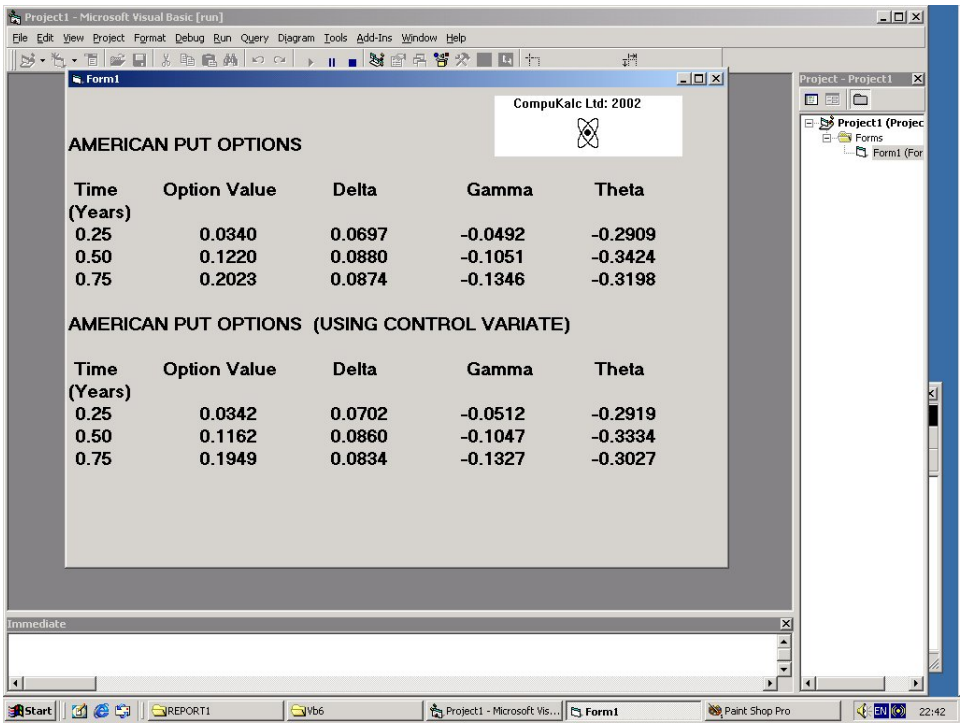

Figure 4: The user form after the derivative calculations have been performed.

3.3 The Optimization Control

This control was created using Visual C++. The control is contained in the file `OPTIM.ocx`, and its instance in this Visual Basic example is called `OPTIM1`.

The example considered here makes use of a nonlinear optimization routine [13],[14], chosen because it is widely used and it has a rather complicated user-interface.It illustrates the use of ActiveX to overcome the inability of the Visual Basic within Excel [15] to supply a user-defined function as an argument to a DLL routine. Mathematical routines that require this feature include those in areas such as Quadrature, Ordinary Differential Equations, Partial Differential Equations, and Optimization.

Rewriting the routine so that it uses reverse communication is another one way that has traditionally been used to overcome the user-defined function argument problem. The COM method outlined here has the advantage that it leaves existing code unchanged and merely provides an "extra layer" of code to support the user-interface.

3.3.1 Language User-Interface

The example considered here was written to perform numerical optimization on the data contained within an Excel spreadsheet. An excerpt from the Visual Basic code is given below.

```
Private Sub Command1_Click()
 Dim x() As Double
 Dim bl() As Double
 Dim bu() As Double
 Dim g() As Double
   .  .  .
```

```
'  For simplicity consider all constraints as nonlinear
   tda = 0
   nclin = 0
   n = 4
   ncnlin = 3
   ReDim x(n)
   ReDim g(n)
   ReDim bl(n + ncnlin)
   ReDim bu(n + ncnlin)

   ' Input the initial values and bounds from the spreadsheet
   For i = 0 To n - 1
      x(i) = Cells(8, 2 + i).Value    ' Initial values for X variables
      bl(i) = Cells(2, 2 + i).Value   ' Lower bounds for X variables
      bu(i) = Cells(3, 2 + i).Value   ' Upper bounds for X variable
   Next i
   ' Input the nonlinear constraints from the spreadsheet
   For i = 0 To ncnlin - 1
      bl(n + i) = Cells(5, 2 + i).Value  ' Lower nonlinear bounds
      bu(n + i) = Cells(6, 2 + i).Value  ' Upper nonlinear bounds
   Next i
' Construct the objective and constraint function names
   objname = "my_objfun"
   constrname = "my_confun"
   full_objname = ActiveWorkbook.Name & "!" & objname
   full_constrname = ActiveWorkbook.Name & "!" & constrname

 ' Set the name of the print function
   OPTIM1.printfun_funname "printit"

' Other optimizer settings
   OPTIM1.List = False      ' Don't list the initial parameter settings
   OPTIM1.obj_deriv = False ' The Jacobian of objective function is not provided
   OPTIM1.con_deriv = False ' The Jacobian of the nonlinear constraint function is
                              ' not provided
' Set the object and constraint function names
   OPTIM1.objfun_funname full_objname
   OPTIM1.confun_funname full_constrname

' Call the optimizer
   OPTIM1.optimize n, nclin, ncnlin, a(0), tda, g(0), x(0), bl(0), bu(0)
   objfun_value = OPTIM1.objf ' get the value of the objective function

' Output the X variable values for the optimal solution
   For i = 0 To n - 1
      Cells(10, 2 + i).Value = x(i)
   Next i

   Cells(11, 2).Value = objfun_value 'Output the optimal value of the objective function

End Sub
```

The control provides methods that permit the user to specify the names of the objective function, the constraint function, and the print function that are to be used in the numerical optimization. Here, the objective function is called `my_objfun`, the constraint function `my_confun`, and the print function `printit`. Figure 5 shows the appearance of the Excel spreadsheet before the numerical optimization has been per-

formed. The variable bounds and the upper and lower constraints for the linear and nonlinear constraint functions can be changed interactively by editing the contents of the appropriate Excel spreadsheet cells. When the command button labelled `Solve` is clicked, the Visual Basic subroutine `Command1_click()` is run and the input data, such as the initial values and the upper and lower constraints, are read from the spreadsheet.

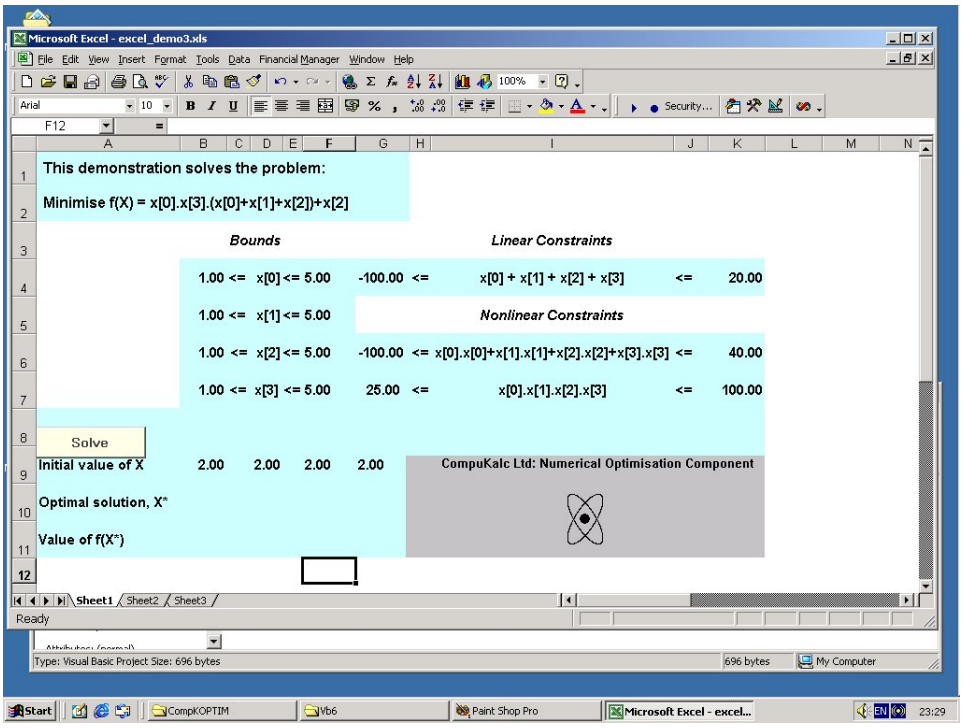

Figure 5: The Excel spreadsheet before the optimization has been performed.

These values are then assigned to internal Visual Basic arrays and are passed to the control `OPTIM1`. The optimization is then performed and the solution vector values, `x(0)...x(3)`, and the optimal value of the objective function are output to the Excel worksheet; see Figure 6.

Here global data and four auxiliary (`ENTRY/EXIT`) housekeeping functions are used to communicate information to the objective and nonlinear constraint functions. The routine `OBJECTIVE_ENTRY` is called just before the call to the objective function and routine `OBJECTIVE_EXIT` just after the call to the objective function. Similarly the function `CONSTRAINT_ENTRY` is called just prior to the call to the nonlinear constraint function and function `CONSTRAINT_EXIT` just after the call to the nonlinear constraint function. This approach allows the objective and constraint functions to be easily defined as follows.

The objective function is

```
Sub my_objfun(num_variables As Long)

  ' The objective function - any valid Visual Basic code is allowed
  ' The optimization control is designed so that the function OBJECTIVE_ENTRY is
  ' called before my_objfun and the function OBJECTIVE_EXIT is called after my_objfun.
  ' This approach reduces the Visual Basic code required in my_objfun to:
```

```
  objective_value = x(0) * x(3) * (x(0) + x(1) + x(2)) + x(2)

End Sub
```

and the user-defined constraint function is

```
Sub my_confun(num_variables As Long, num_constraints As Long)

  ' The nonlinear constraint function - any valid Visual Basic code is allowed
  ' The optimization control is designed so that the function CONSTRAINT_ENTRY is
  ' called before my_confun and the function CONSTRAINT_EXIT is called after my_confun.
  ' This approach reduces the Visual Basic code required in my_confun to:

   constraint_value(0) = x(0) + x(1) + x(2) + x(num_variables - 1)
   constraint_value(1) = x(0) * x(0) + x(1) * x(1) + x(2) * x(2) + x(3) * x(3)
   constraint_value(2) = x(0) * x(1) * x(2) * x(num_variables - 1)

End Sub
```

The user-defined print function is considered in the following section.

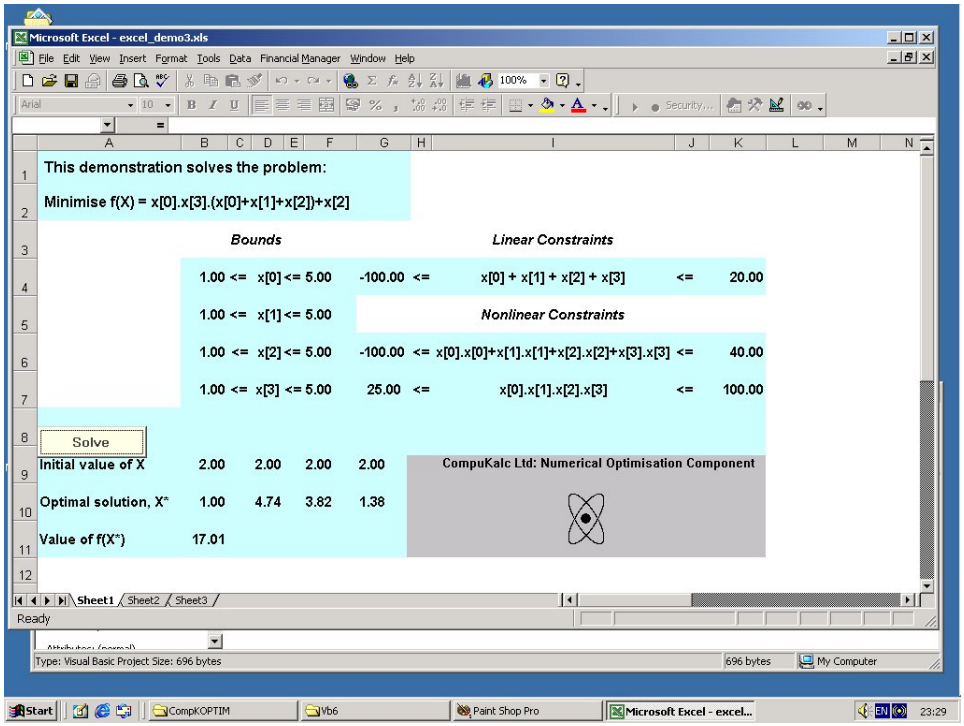

Figure 6: The Excel spreadsheet after the optimization has been performed.

3.3.2 User-defined Print Function

Monitoring how an optimization program is proceeding can be a problem from within Windows. The usual approach is to write the output to a file for later display. However, this method does not allow the user to display the monitoring information programmatically.

In this example, the ActiveX control `OPTIM1` has been constructed to use the Excel Vi-

sual Basic subroutine printit for the output of monitoring information. The subroutine printit has twelve arguments and these can be used to display monitoring information such as the number of iterations, the value of the objective function, the intermediate solution vectors, etc. The user has the flexibility of writing any valid Visual Basic code within printit to display these values. Figure 7 shows the monitoring information for this example. Here all the information was displayed within Excel Sheet3 and was output in the following order:

• The major iteration count.

• The number of minor iterations required by the feasibility and optimality phases of the QP subproblem.

• The step taken along the computed search direction. On reasonably well behaved problems the unit step will be taken as the solution is approached.

• The intermediate solution vector.

• The value of the augmented Lagrangian merit function at the current iterate. This will usually decrease at each iteration. As the solution is approached it will converge to the value of the objective function at the solution.

• The Euclidean norm of the projected gradient. This will be approximately zero in the neighbourhood of a solution.

The print function is as follows:

```
Sub printit(n As Long, it_maj_prt As Long, sol_prt As Long, maj As Long, mnr As Long, _
   step As Double, nfun As Long, merit As Double, violtn As Double, norm_gz As Double,_
    cond_hz As Double, x_ptr As Long)

   ' The user-defined print function. The user can decide the format in which any of
   ' the twelve arguments to printit are to be output.

   Dim xp() As Double
   ReDim xp(n)
   Call get_darray(x_ptr, xp(0), n) ' Get the X variable values
   With Sheets("Sheet3")
   If Row = 1 Then ' First call so output the headers
        Row = Row + 2
        .Cells(Row, 1).Value = "  Major Iter"
        .Cells(Row, 2).Value = "  Minor Iter"
        .Cells(Row, 3).Value = "Step"
         For i = 0 To n - 1
          .Cells(Row, 4 + i).Value = "x[" & i & "]"
         Next i
        .Cells(Row, n + 4).Value = "Merit Fn"
        .Cells(Row, n + 5).Value = "Norm Gz"
        Row = Row + 2
     End If
     If (it_maj_prt) Then ' A major iteration
       .Cells(Row, 1).Value = maj ' The major iteration count
       .Cells(Row, 2).Value = mnr ' The number of minor iterations of the QP subproblem
       .Cells(Row, 3).Value = Format(step, "0.00E+00") ' The step length along the
```

```
                                                        ' search direction
        For i = 0 To n - 1 ' Output the current X variable values
         .Cells(Row, 4 + i).Value = Format(xp(i), "##.00")
        Next i
    ' Output the value of the augmented Lagrangian merit function at the current
    ' point. As the solution is approached this should converge to the optimal value
    ' objective function

       .Cells(Row, n + 4).Value = Format(merit, "0.00E+00")
    ' Output the Euclidean norm of the projected gradient. This should be approximately
    ' zero when the X variables have their optimal values.

       .Cells(Row, n + 5).Value = Format(norm_gz, "0.00E+00")
       Row = Row + 1 ' Increment the row count by one, for the next output line
     End If
   End With
End Sub
```

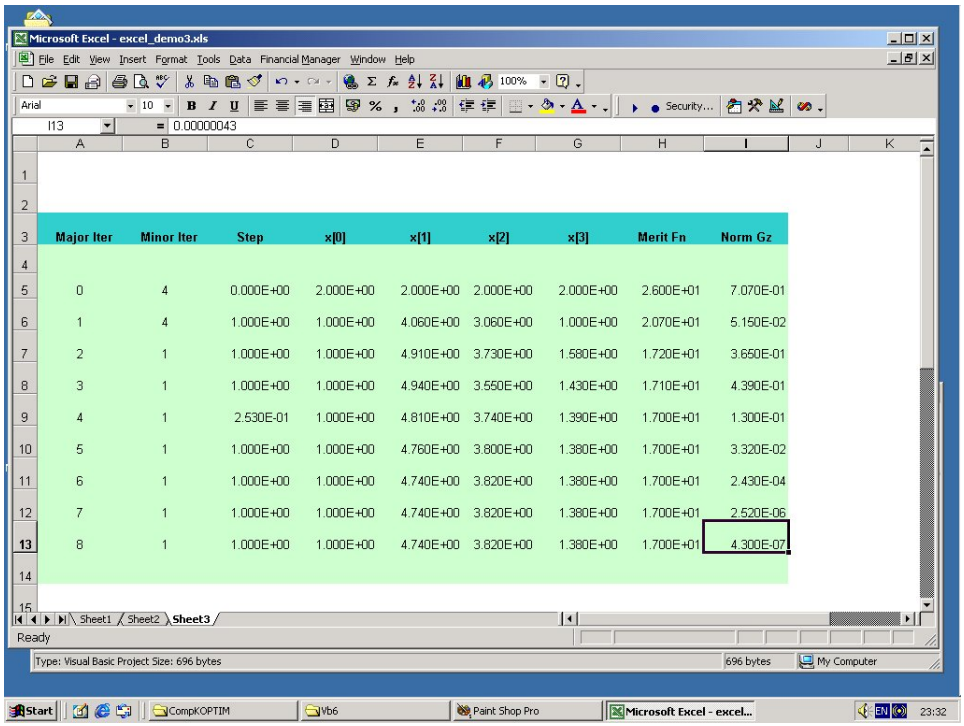

Figure 7: An Excel spreadsheet with intermediate optimization output.

Although this example is fairly simple it can be extended, through the inclusion of more variables and customised objective and constraint functions, to solve optimization problems that are of great practical interest. Current areas where this might be of significant benefit include financial portfolio optimization [16] and the modelling of financial time series using GARCH [17]. It should be mentioned that although the the method outlined here is slower than if all the computation had been done in C++ it still gives *reasonable* performance. For instance (using a 800MHZ PC) it was found that problems involving the optimization of 200 variables took about 5-10 secs to compute.

4 Inprise Delphi

The purpose of this section is to show how numeric ActiveX components can be incorporated into Inprise Delphi applications. Since Delphi is similar to Visual Basic only brief details will be given.

4.1 Derivative Pricing Control

Excerpts from the Delphi source code are given below.

```
procedure TForm1.FormClick(Sender: TObject);
var
  greeks: Array[1..5] of double;
  T: double;
  i: integer;
  opt_val: double;
  num_precision: integer;
  num_digits: integer;
  pos: integer;
  val1: String;
begin
   DERIV2.putcall := 1;       {A put option}
   DERIV2.curval := 10.0;     {The current value of the asset}
   DERIV2.strike := 8.0;      {The strike price for the option}
   DERIV2.dividends := 0.06;  {The continuous dividend yield}
        .    .    .
   Canvas.TextOut(10,80,'AMERICAN PUT OPTIONS (USING CONTROL VARIATE)');

   Canvas.TextOut(10,140,'Time');
         .    .    .
   for i := 1 To 3 Do
      Begin
        T := i*0.25;
         .    .    .
        DERIV2.maturity := T;            {Set the maturity of the option, in years}
        DERIV2.Calculate;                {Do the calculation}
        DERIV2.greeks(greeks[1]);        {Get the hedge statistics, the greeks}
        opt_val := DERIV2.optval;        {Get the option value}
        val1 := FloatToStrF(T,ffFixed,num_precision,num_digits);
        Canvas.TextOut(10,pos,val1);
        val1 := FloatToStrF(opt_val,ffFixed,num_precision,num_digits);
        Canvas.TextOut(100,pos,val1);
        val1 := FloatToStrF(greeks[1],ffFixed,num_precision,num_digits);
         .   .   .
End;
end;

end;
```

4.2 Optimization Control

This section illustrates the use of an ActiveX optimization component to construct a Delphi application with a similar user-interface to the Excel example described in Section 3.3.The differences between these two examples include the following:

• The optimization component used here does not have a user-defined print function.

• There are no (ENTRY/EXIT) housekeeping routines as in the Excel example.

• All data input and data output is performed using Delphi textbox components.

• All explanatory text is placed within Delphi label components.

Excerpts from the Delphi source code are given below.

```
procedure TForm1.SOLVEClick(Sender: TObject);

var
     .  .  .
 num_digits:integer;
 val1:String;
 obj_val:double;
begin

 {For simplicity consider all the constraints as nonlinear}
   n := 4;
   tda := 0;
   nclin := 0;
   ncnlin := 3;
   num_vars := n;
   num_cons := ncnlin;

   {Input the initial values for the X variables}
   loc_x[0] := StrToFloat(XIN1.Text);
   loc_x[1] := StrToFloat(XIN2.Text);
   loc_x[2] := StrToFloat(XIN3.Text);
   loc_x[3] := StrToFloat(XIN4.Text);

   {Input the lower bounds of the X variables}
   bl[0] := StrToFloat(B1L.Text);
   bl[1] := StrToFloat(B2L.Text);
   bl[2] := StrToFloat(B3L.Text);
   bl[3] := StrToFloat(B4L.Text);
    .     .     .
   {Do the optimization}
   OPTD21.optimize(n,nclin,ncnlin,a[0],tda,g[0],loc_x[0],bl[0],bu[0]);
   obj_val := OPTD21.Objval; {get the value of the objective function}
       .    .    .
   {Output the optimal value of the X variables}
   XOUT1.Text := FloatToStrF(loc_x[0],ffFixed,num_precision,num_digits);
   XOUT2.Text := FloatToStrF(loc_x[1],ffFixed,num_precision,num_digits);
   XOUT3.Text := FloatToStrF(loc_x[2],ffFixed,num_precision,num_digits);
   XOUT4.Text := FloatToStrF(loc_x[3],ffFixed,num_precision,num_digits);

   {Output the optimal value of the objective function}
   OBJVAL.Text := FloatToStrF(obj_val,ffFixed,num_precision,num_digits);

end;

procedure TForm1.OPTD21Objfunction(Sender: TObject);
var
```

```
 obj_val:double;

begin
 {The objective function}

  OPTD21.getvars(x[0],num_vars);             {Get the X variable values}

  obj_val := x[0]*x[3]*(x[0]+x[1]+x[2])+x[2];  {Set the value of the objective function}

  OPTD21.setvars(x[0],num_vars);            {Set the X variable values}
  OPTD21.Objval := obj_val;                 {Set the value of the objective function}
end;

procedure TForm1.OPTD21Constrfunction(Sender: TObject);
begin
  {The nonlinear constraint function}

  OPTD21.getvars(x[0],num_vars);                    {Get the X variable values}
  OPTD21.getconstr(constraint_value[0],num_cons);   {Get the constraint values}

  {Set the value of the nonlinear constraint function}

  constraint_value[0] := x[0]+x[1]+x[2]+x[num_vars-1];
  constraint_value[1] := x[0]*x[0]+x[1]*x[1]+x[2]*x[2]+x[3]*x[3];
  constraint_value[2] := x[0]*x[1]*x[2]*x[num_vars-1];

  OPTD21.setconstr(constraint_value[0],num_cons);   {Set the constraint values}
  OPTD21.setvars(x[0],num_vars);                    {Set the X variable values}

end;
```

5 Conclusions

This paper has discussed the use of numeric ActiveX component software with primitive visual user-interfaces. It has demonstrated that these software components are flexible and can easily be incorporated into (and removed from) existing applications. These (usually invisible at run-time) numeric components are therefore ideal computational engines for applications that have sophisticated user-interfaces.

The paper has given results concerning an ActiveX optimization which allows the user to define both the objective and constraint functions with Visual Basic. Although this method is slower than using straight C++ it was found to give *reasonable performance*. For instance (using a 800MHZ PC) an Excel optimization problem involving 200 variables was found to take 5-10 secs to compute.

Since ActiveX component technology is based on C++, calls to complicated routines can be simplified through the use of properties, methods, events, object initialization via constructors, data/information hiding within the object, and also optional arguments that take default values.

ActiveX components can be used by the entire range of Microsoft products, from PowerPoint to Internet Web browsers, and also by other Windows products such as Inprise Delphi.

To summarise, some of the advantages of using numeric ActiveX library components are:

• They can easily be incorporated into the complete range of Microsoft products and other Windows software such as Inprise Delphi.

• They provide interactive help information concerning their properties, methods and events through the use of a type library.

• They allow the creation of simple user-interfaces to complicated routines.

Possible factors against creating a large library of ActiveX components are

• The library is a PC Windows based product and so cannot be used within UNIX.

• The manpower required to create such a product would be much greater than that for existing numeric DLLs.

However, the overall benefits to be gained from using numeric ActiveX components suggest they will be increasingly used for mathematical computing.

8 References

[1] G F Levy, Mathemantics, Visual Systems Journal, 3, 28-36, 1997

[2] G F Levy, Mathemantics part II, Visual Systems Journal, 4, 26-35, 1997

[3] A Denning, ActiveX Controls Inside Out, Microsoft Press, 1997

[4] K Brockshmidt, Inside OLE, Microsoft Press, 1995

[5] D Rogerson, Inside COM, Microsoft Press, 1997

[6] D Box, Essential COM, Addison Wesley, 1998

[7] D Kruglinski, G Shepherd and S Wingo, Programming Microsoft Visual C++, Microsoft Press, 1998

[8] E O Brigham, The Fast Fourier Transform, Prentice-Hall, 1973

[9] J Hull, Options Futures and Other Derivatives, Prentice Hall, 1997

[10] J C Cox, S A Ross and M Rubinstein, Option Pricing: A Simplified Approach, Journal of Financial Economics 7, 229 - 263, 1979

[11] M Broadie and J DeTemple, American Option Valuation: New Bounds, Approximations, and a Comparison of Existings Methods, The Review of Financial Studies 9, No 4, 1211-1250, 1996

[12] F Black and M Scholes, The Pricing of Corporate Liabilities, Journal of Political Economy 81 637-657, 1973

[13] P E Gill, W Murray and W H Wright, Practical Optimization, Academic Press, 1981

[14] B A Murtagh and M A Saunders, MINOS 5.4 User's Guide Report SOL 83-20R Department of Operations Research, Stanford University, 1995.

[15] Excel/Visual Basic Programmers Guide, Microsoft Corporation, 1995

[16] H M Markowitz, Mean Variance Analysis in Portfolio Choice and Capital Markets, Basil Blackwell, 1989

[17] T Bollerslev, Generalized Autoregressive Conditional Heteroskedasticity, Journal of Econometrics 31, 307-27, 1986