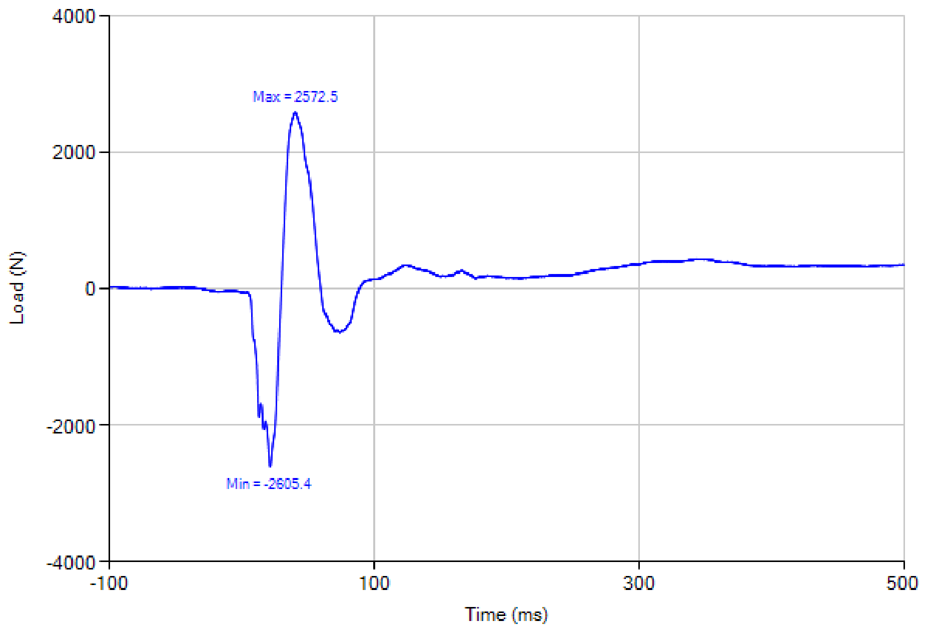
An MSChart Class for Graphing Line Series

Darryl Bryk
U.S. Army RDECOM-TARDEC
Warren, MI 48397



Introduction

This article will describe a C# class for utilizing the Microsoft chart control to graph data as a line series. The MSChart control provided in .NET is quite versatile. The Microsoft Office products also utilize a version of the control for graphing data, which can be accessed using VBA code. Others have written about some of the basics of using the MSChart control for various types of graph types, but when I was looking into graphing line series, there was not much help out there, so I decided to write this article.

Using the Code

The class variables and constructor for the class CGraph are shown below:

```
//-----------------------------------------------------------------------------
// Class CGraph
//-----------------------------------------------------------------------------
public class CGraph {
    public TabPage Tab { get; set; }

    private Chart chart;
    private ChartArea chArea;

//-----------------------------------------------------------------------------
// Class constructor
//-----------------------------------------------------------------------------
public CGraph(TabPage Contr) {
    chart = new Chart();
    Contr.Controls.Add(chart); // Add chart to control
    Tab = Contr;
    chart.Dock = DockStyle.Fill; // Size to the control
    chart.GetToolTipText += new EventHandler<ToolTipEventArgs>(this.chart_ToolTip);

    Legend leg = chart.Legends.Add(null);
    leg.Docking = Docking.Bottom;
    leg.Font = new Font(chart.Legends[0].Font.FontFamily, 8); // Font size

    chArea = chart.ChartAreas.Add(null);

    chart.Palette = ChartColorPalette.None;
    chart.PaletteCustomColors = new Color[] {Color.Blue, Color.DarkRed,
                                            Color.Green, Color.Orange,
                                            Color.SteelBlue, Color.Magenta,
                                            Color.Purple, Color.OliveDrab,
                                            Color.DodgerBlue, Color.Sienna,
                                            Color.Teal, Color.YellowGreen,
                                            Color.RoyalBlue, Color.Black,
                                            Color.Teal, Color.OrangeRed};

    // Enable range selection and zooming
    chArea.CursorX.IsUserEnabled = true;
    chArea.CursorX.IsUserSelectionEnabled = true;
    chArea.AxisX.ScaleView.Zoomable = true;
    chArea.AxisX.ScrollBar.IsPositionedInside = true;
    chArea.AxisX.ScaleView.SmallScrollSize = 1;
    chArea.AxisX.IntervalAutoMode = IntervalAutoMode.VariableCount;
    chArea.AxisY.IntervalAutoMode = IntervalAutoMode.FixedCount;

    chArea.AxisX.LabelStyle.Format = "F0";
    chArea.AxisX.LabelStyle.IsEndLabelVisible = true;

    chArea.AxisX.MajorGrid.LineColor = Color.FromArgb(200, 200, 200);
    chArea.AxisY.MajorGrid.LineColor = Color.FromArgb(200, 200, 200);
    chArea.AxisX.LabelStyle.Font =
        new Font(chArea.AxisX.LabelStyle.Font.Name, 9, FontStyle.Regular);
    chArea.AxisY.LabelStyle.Font =
        new Font(chArea.AxisY.LabelStyle.Font.Name, 9, FontStyle.Regular);
    chArea.AxisX.TitleFont =
                new Font(chArea.AxisX.TitleFont.Name, 9, FontStyle.Regular);
    chArea.AxisY.TitleFont =
                new Font(chArea.AxisY.TitleFont.Name, 9, FontStyle.Regular);
}
```

The constructor gets a `TabPage` control passed to it to add the new chart onto, but this could be easily changed to another type of control, or made generic as an object. The control is saved to a public class variable named `Tab` so other methods have access. The new chart object is added and saved in the private class variable called `chart`. An event handler is defined to handle ToolTips for the chart called `chart_ToolTip` (explained further below). The legend is added and docked to the bottom of the chart, and its font is set. In addition to the chart declaration, a chart area is required, and this is added to the chart and saved for the class as variable `chArea`. The default MSChart graphing palette is changed to custom colors by calling `PaletteCustomColors`. This is not necessary as the chart control has its own palette defaults, but this shows how custom colors can be defined. The first series in the chart will use the first palette color, the second will use the second color, etc.

The next several lines setup x-axis zooming for the graph. The y-axis may also be set for zooming in a similar way, but was not done for this application. This is a nice feature for a line series graph and is automatically handled by the control. If the user clicks and drags an area on the graph it will be automatically zoomed and a scrollbar added for positioning. The user can even zoom again to get a higher resolution view if needed. A small button on the scrollbar resets the graph to the previous zoom level. Each click of the button takes the graph one level back until the original un-zoomed graph is restored (like an undo). The chart's x and y axis intervals are set which determine how many axis labels and "tic marks" are drawn. The remaining lines set label styles, fonts, and grid colors to a grey level.

The `Title` method shown below adds the main title to the chart in the specified color and can be docked to various defined chart positions (top, bottom, etc.). It first checks to see if the title string passed to it has been used already in the same docking position on the chart. If it has been used in the same docking position the method returns without adding it to the chart, otherwise, the title is appended to the current title, or if there were no titles in that docking position, it adds it. This is a useful feature because the calling function may have the option to graph multiple data series to the same chart, and may for example, add the series name as it encounters it by looping, and so this method will add each new title and also ensures that a title line won't get duplicated. The line that changes the title font is not used here, but is shown commented out to show how it can be done. Adding x-axis and y-axis titles are done similarly with functions provided in the class.

```
//-----------------------------------------------------------------------------
// Add graph title. Repeated calls append new titles if unique or if new pos.
//-----------------------------------------------------------------------------
public void Title(string title, Docking position, Color Colr) {
    foreach (Title t in chart.Titles) {
        if (t.Docking == position) {
            if (!t.Text.Contains(title)) t.Text += "\n" + title; // Append
            return;
        }
    }

    // If here add new title
    Title newtitle = new Title(title, position);
    chart.Titles.Add(newtitle);
    newtitle.ForeColor = Colr;
    //newtitle.Font = new Font(newtitle.Font.Name, 8, FontStyle.Bold);
}
```

The method `GraphLineSeries`, shown below, does the actual line series graph onto the chart. This function may be called multiple times to add more than one data series to the same chart. The function is passed the series name, the x and y data arrays, a Boolean that tells whether to add min/max labels, the series graph color (in case one wants to override the default), and another Boolean to add the legend or not for the series. The function returns the successfully added series object or null if a series could not be added.

The new series is declared with the passed string `Name`. The code that adds the series to the chart is enclosed in a `try` block because the call to add the series will throw an exception if the series name has already been used in the chart. This is useful to "trap" redundant calls with the same series name so that the caller doesn't have to check this. If the series name was already used the function returns null without graphing anything. For a line series there must be a single dimensional array of both x values and y values. The data arrays are attached to the series and drawn to the graph with a simple call to `DataBindXY(x, y)`. Note that a chart control can contain more than one chart area. The if statement checking the ChartAreas.Count is in case there is more than one chart area on the chart control, in which case, the data is bound to the last chart area. Normally I use only one chart area per control, but the provision is there if needed. If a graph legend is desired the Boolean `inLegend` should be passed as true, otherwise the legend is not added. The series color can be explicitly set from the passed variable `colr`, or if `colr` is passed as null, the default colors defined in the constructor are used.

```
//-----------------------------------------------------------------------------
// Adds line series to chart.
//-----------------------------------------------------------------------------
public Series GraphLineSeries(string Name, double[] x, double[] y,
                              bool lblminmax, Color colr, bool inLegend) {
    Series ser = new Series(Name);

    try { // Traps redundant series names
        chart.Series.Add(ser);
        ser.ChartType = SeriesChartType.Line;
        if (chart.ChartAreas.Count() > 1) // Bind to last chart area
            ser.ChartArea = chart.ChartAreas[chart.ChartAreas.Count() - 1].Name;
        ser.Points.DataBindXY(x, y);
    }
    catch (ArgumentException) { return null; } // Handle redundant series
    catch (OutOfMemoryException err) {
        MessageBox.Show("Error - " + err.Message, "GraphLineSeries()",
                        MessageBoxButtons.OK, MessageBoxIcon.Error);
        return null;
    }

    ser.IsVisibleInLegend = inLegend;
    if (colr != null) ser.Color = colr;

    if (lblminmax) { // Add min, max labels
        chart.ApplyPaletteColors(); // Force color assign so labels match
        SmartLabels(ser, ser.Color, ser.Color, FontStyle.Regular);
        ser.SmartLabelStyle.MovingDirection = LabelAlignmentStyles.TopLeft;

        DataPoint p = ser.Points.FindMaxByValue();
        p.Label = "Max = " + p.YValues[0].ToString("F1");
        p = ser.Points.FindMinByValue();
        p.Label = "Min = " + p.YValues[0].ToString("F1");
    }

    return ser;
}
```

If the passed Boolean `lblminmax` is true then minimum and maximum data values will be labeled on the chart. The label color is set to the same color as the series to easily identify it in case more than one series is graphed to the same chart. Note that `ApplyPaletteColors()` must be called here so the palette change defined in the constructor is used for the labels. The MSChart methods `FindMaxByValue()` and `FindMinByValue()` are called to find the index position of the maximum and minimum data values, respectively, so the labels can be assigned to the corresponding data point.

The function `SmartLabels()` is called (shown below) to handle the SmartLabel settings for the class. The MSChart SmartLabels function automatically repositions labels on the graph so that they don't interfere with other graph labels. SmartLabels also has a "callout line" feature which draws a line segment from the data point to the label if needed. This helps to identify the labeled min/max point for each series on the chart when the points or labels lie close to each other. The call to `SmartLabelStyle.MovingDirection`, which defines how the labels are to be directed away from the points, is not included in the class's `SmartLabels` method in order to give control to outside functions which may need to define the moving direction based on some variable's value, e.g. if positive, move above, or if negative, move below.

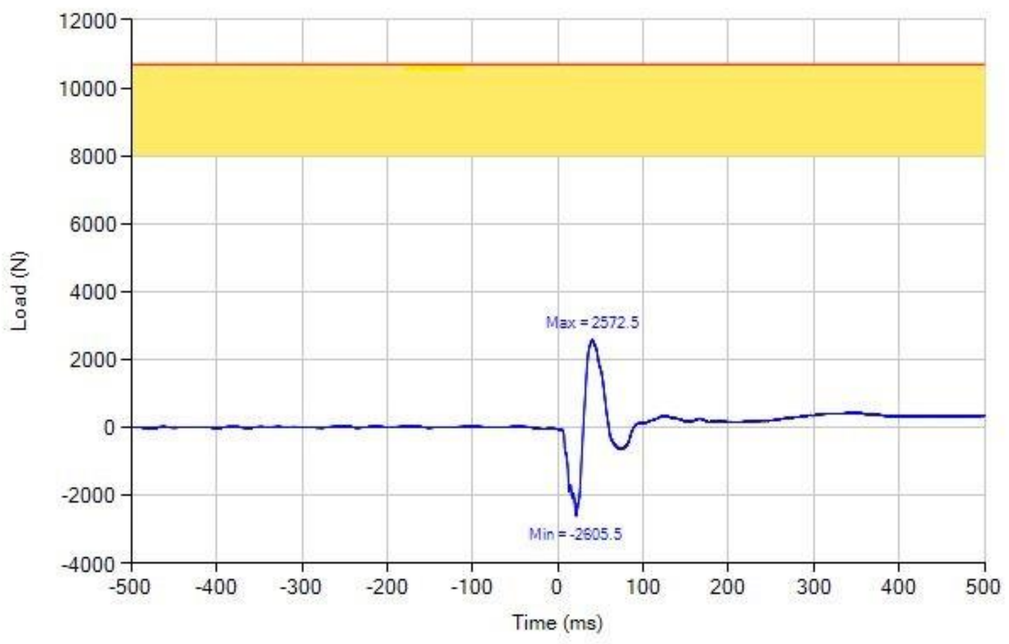
`SmartLabels` takes the series object as input, and attributes for the label's font, color, and "callout" line color. Further information on the function calls used here for setting up SmartLabels can be found in the Microsoft documentation.

```
//----------------------------------------------------------------------------
// Setup Smart Labels
//----------------------------------------------------------------------------
private void SmartLabels(Series ser, Color Fcolor, Color Linecolor,
                         FontStyle FontStyle, int FontSize = 7) {
    if (ser == null) return;

    ser.LabelForeColor = Fcolor;
    ser.Font = new Font(ser.Font.Name, FontSize, FontStyle);
    ser.SmartLabelStyle.Enabled = true;
    ser.SmartLabelStyle.CalloutLineColor = Linecolor;
    ser.SmartLabelStyle.CalloutStyle = LabelCalloutStyle.None;
    ser.SmartLabelStyle.IsMarkerOverlappingAllowed = false;
    ser.SmartLabelStyle.MinMovingDistance = 1;
    ser.SmartLabelStyle.IsOverlappedHidden = false;
    ser.SmartLabelStyle.AllowOutsidePlotArea = LabelOutsidePlotAreaStyle.No;
    ser.SmartLabelStyle.CalloutLineAnchorCapStyle = LineAnchorCapStyle.None;
}
```

The code shown below is the ToolTip handler for the class. If the user hovers the mouse over a data point in the graph this ToolTip handler is fired (from the event handler defined in the constructor) and shows the series name and the x and y data values at that point. ToolTips can be added to every data point, but this adds considerable overhead to the chart rendering if there are a lot of data points in a series. Adding the ToolTips by this mouse interrupt, only when hovering, enables them only on demand, so is less costly in time. The method also provides a couple ToolTip helps when hovering over the axis and axis labels area, and over the scrollbar that the chart adds when a zoom has been done, telling the user about the click-drag for zoom and zoom undo features.

```
//---------------------------------------------------------------------------
// Chart tooltip handler.
//---------------------------------------------------------------------------
private void chart_ToolTip(object sender, ToolTipEventArgs e) {
    HitTestResult h = e.HitTestResult;

    switch (h.ChartElementType) {
        case ChartElementType.Axis:
        case ChartElementType.AxisLabels:
            e.Text = "Click-drag in graph area to zoom";
            break;
        case ChartElementType.ScrollBarZoomReset:
            e.Text = "Zoom undo";
            break;
        case ChartElementType.DataPoint:
            e.Text = h.Series.Name + '\n' + h.Series.Points[h.PointIndex];
            break;
    }
}
```

The method below I will include because it may be useful for graphing range type graphs and not much is written on these graph types. It allows you to graph a line segment in calculated widths. This differs from just making broad line segments by using a wider "pen", because the desired height of the line could be other than some integer multiple of a pen width. An example graph is shown below where the range series is in yellow denoting a peak range level. The red line was added to show where the maximum level is by using `GraphLineSeries()`.



The `GraphRangeSeries()` method looks pretty much like `GraphLineSeries()` except requires the height array which is passed in the variable `yht`, which defines the graphed height of the series (i.e. y – yht is the height). Note that the call to `DataBindXY` takes this third array as an argument. The graphed series is automatically filled in with the color specified.

```
//----------------------------------------------------------------------------
// Adds Range type series to chart.
//----------------------------------------------------------------------------
public Series GraphRangeSeries(string sername, double[] x, double[] y,
                               double[] yht, Color colr) {
    System.Windows.Forms.Cursor.Current = Cursors.WaitCursor;
    Series ser = new Series(sername);

    try {
        chart.Series.Add(ser);
        if (chart.ChartAreas.Count() > 1) // Bind to last chart area
            ser.ChartArea = chart.ChartAreas[chart.ChartAreas.Count() - 1].Name;
        ser.ChartType = SeriesChartType.Range;
        ser.IsVisibleInLegend = false;
        ser.Color = colr;

        ser.Points.DataBindXY(x, y, yht);
    }
    catch (ArgumentException) { return null; } // Handle redundant series
    catch (Exception err) {
        MessageBox.Show("Error - " + err.Message, "GraphRangeSeries()",
                        MessageBoxButtons.OK, MessageBoxIcon.Error);
        return null;
    }

    System.Windows.Forms.Cursor.Current = Cursors.Default;

    return ser;
}
```

Conclusion

This class is by no means exhaustive of the MSChart control’s functionality, but will hopefully offer the developer a good starting point for graphing data as a line series.