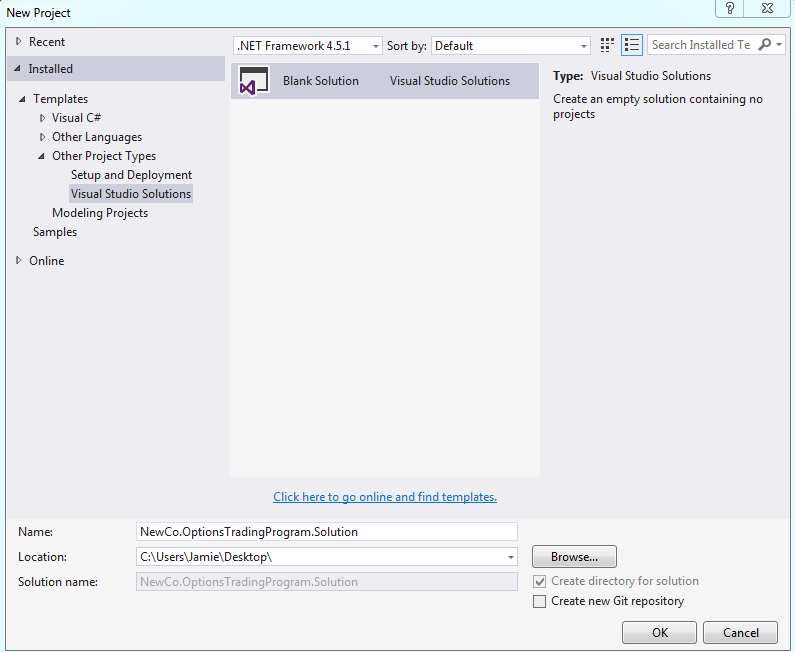
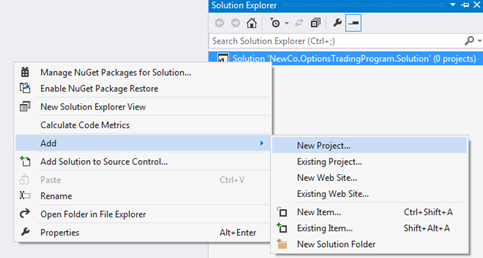
**Set Up The Solution**

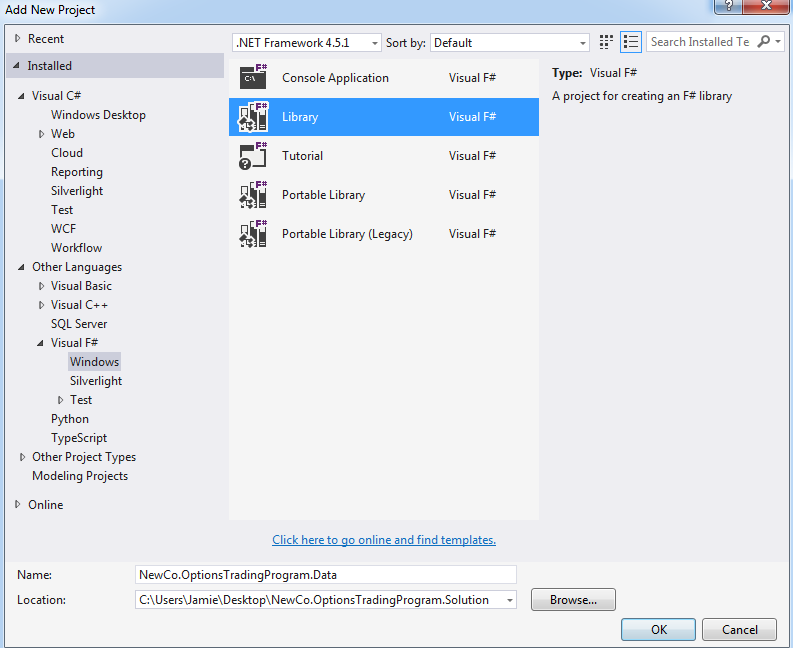
Open Visual Studio

Create a new solution (File -> New -> Project) called NewCo.OptionsTradingProgram.Solution

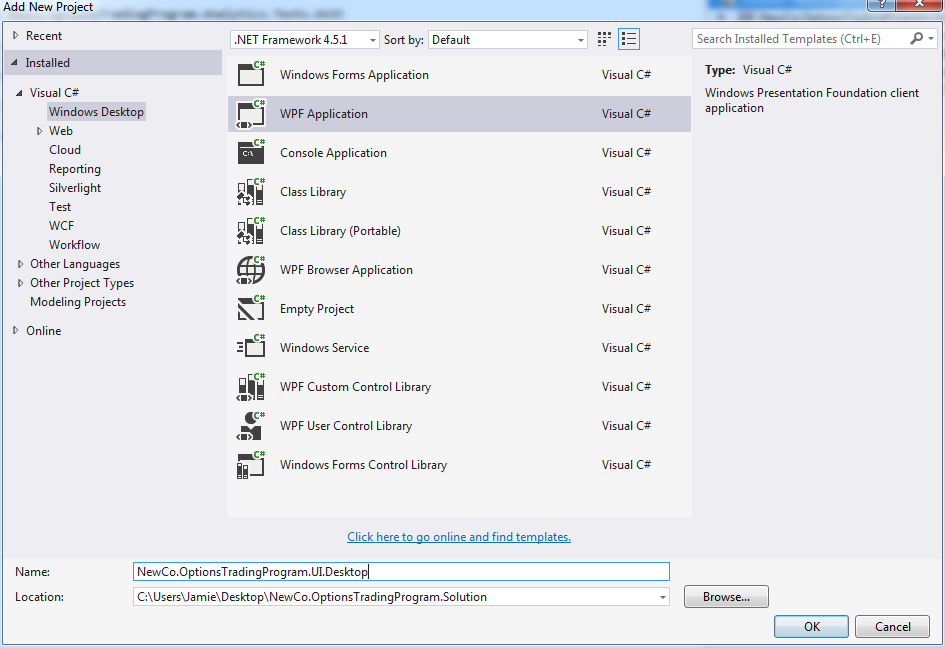


In the solution explorer, add a new project (Solution Explorer -> Right Click -> New Project) and select Visual F# -> Windows -> Library project. Call it NewCo.OptionsTradingProgram.Data

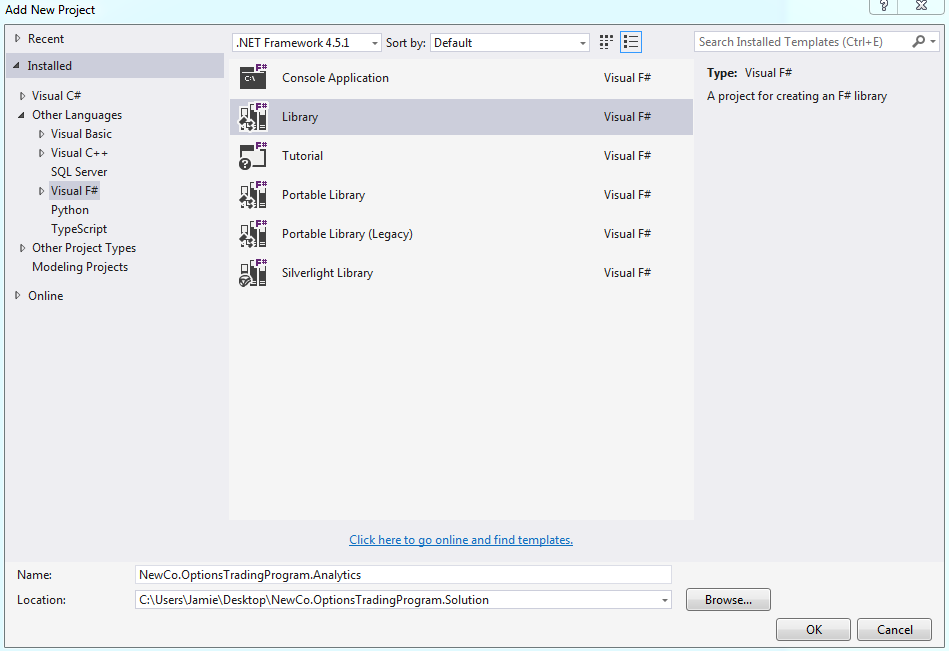




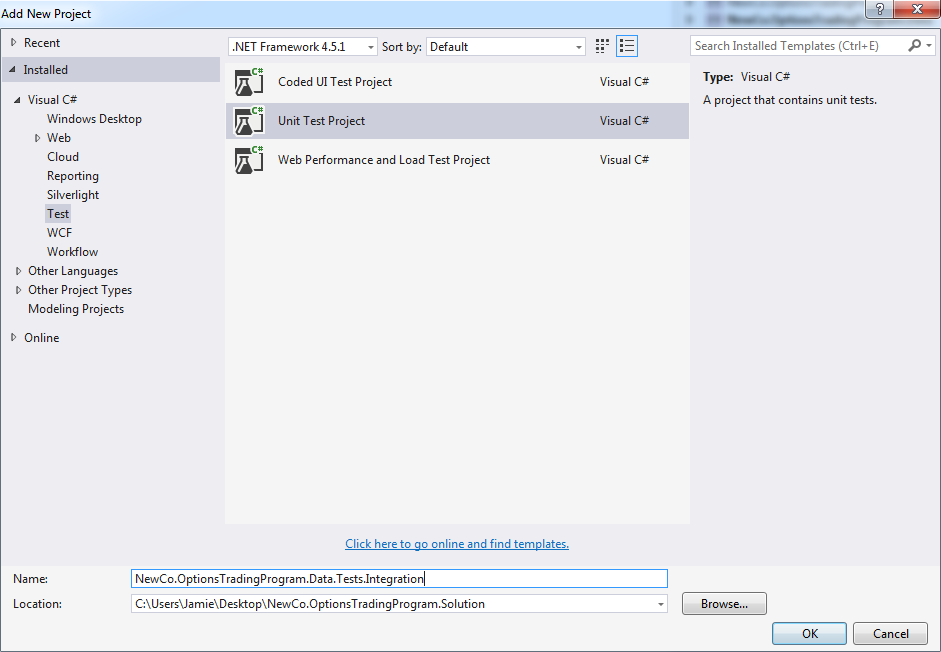
In the solution explorer, add a new project (Solution Explorer -> Right Click -> New Project) and select Visual C# -> Windows Desktop -> WPF Application. Call it NewCo.OptionsTradingProgram.UI.Desktop



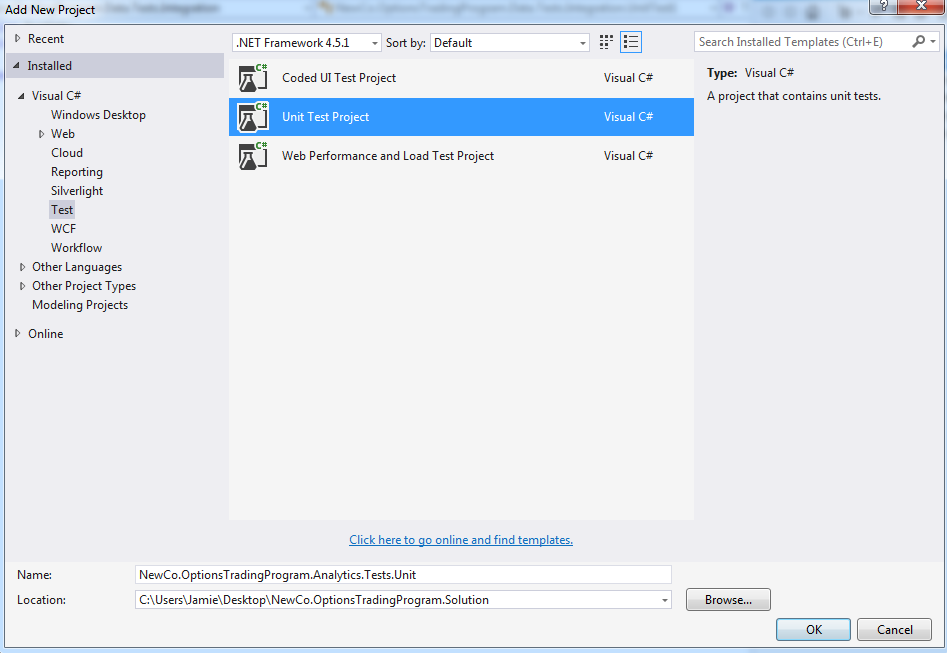
In the solution explorer, add a new project (Solution Explorer -> Right Click -> New Project) and select Visual F# -> Windows -> Library project. Call it NewCo.OptionsTradingProgram.Analytics



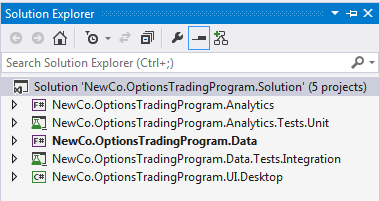
In the solution explorer, add a new project (Solution Explorer -> Right Click -> New Project) and select Visual C# -> Test -> Unit Test Project. Call it NewCo.OptionsTradingProgram.Data.Tests.Integration.



In the solution explorer, add a new project (Solution Explorer -> Right Click -> New Project) and select Visual C# -> Test -> Unit Test Project. Call it NewCo.OptionsTradingProgram.Analytics.Tests.Unit.

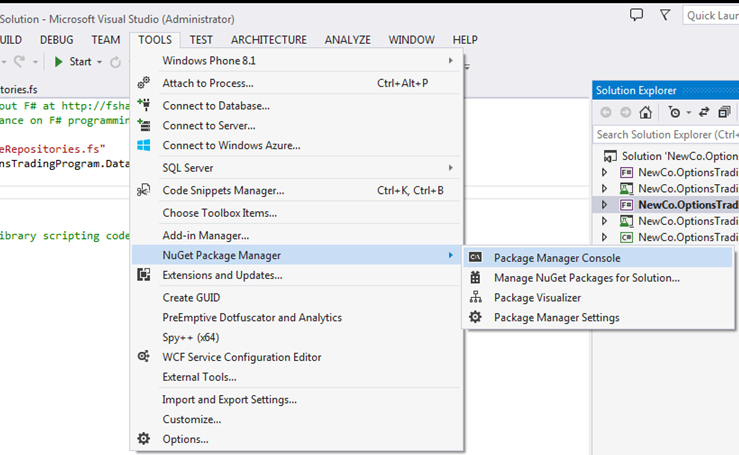


Your solution explorer should look like this

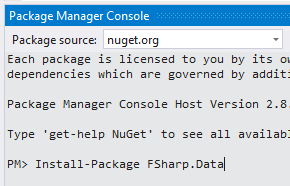


**Access The Data**

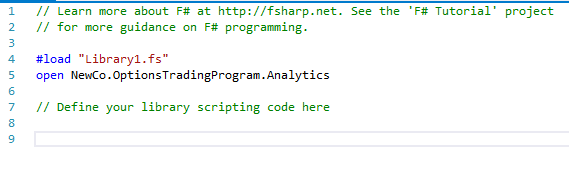
Open the NewCo.OptionsTradingProgram.Data project and open the NuGet package Manager Console (TOOLS -> NuGetPackageManager->Package Manager Console)



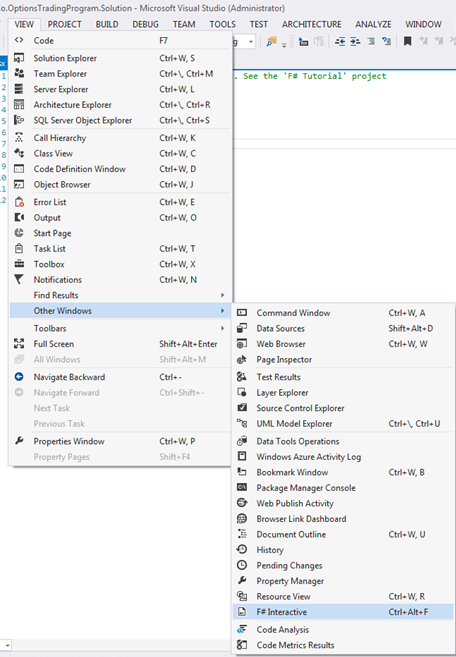
At the command prompt, enter the following command:



Open the script.fsx file.



Open the F# Interactive (the built-in REPL)



We are now ready to “prove out” data access from Yahoo.

In the Script.fsx file, remove all code and type in the following bock.

#r "C:\Users\Jamie\Desktop\NewCo.OptionsTradingProgram.Solution\packages\FSharp.Data.2.0.8\lib\portable-net40+sl5+wp8+win8\FSharp.Data.dll"

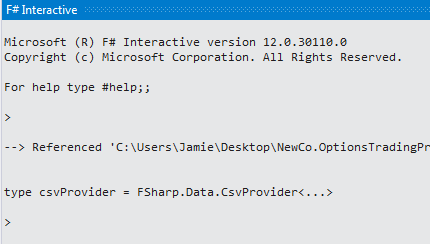
open FSharp.Data

type csvProvider = CsvProvider<"http://ichart.finance.yahoo.com/table.csv?s=MSFT">

Note that the actual path will be different for your machine (wherever Nuget stuck the file). You can open the folder via solution explorer and copy/paste the path so you don’t have to type the path

Once you finish with the #r line, you will get a permission dialog box, Hit “enable”.

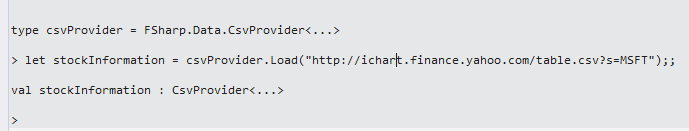
Highlight all 4 lines and hit ALT+ENTER. In your FSI, you should see:



We are going to now inspect the data in the REPL. At the “>”, enter the following line:

let stockInformation = csvProvider.Load("http://ichart.finance.yahoo.com/table.csv?s=MSFT");;

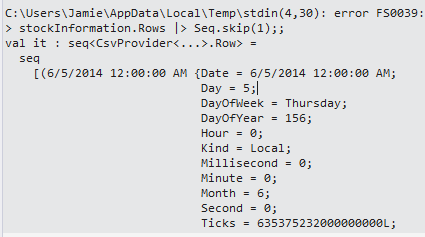
Note that there is a two semi-colons in the REPL. Your REPL should look like this:



Next, type this:

stockInformation.Rows |> Seq.Skip(1);;

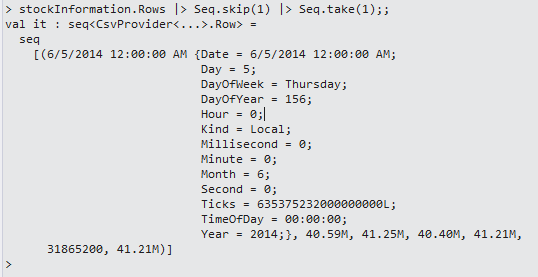
You should get something like this:



Then, type this into the REPL:

stockInformation.Rows |> Seq.skip(1) |> Seq.take(1);;

And you should get something like this:



Which is the 2nd row of data coming back from Yahoo. If you want to see the column headers, type this into the REPL (Note that you can hit the up arrow if you don’t want to type the same commands in again):

stockInformation.Headers;;

And you should get something like this:



With our concept ‘proved out’ in the REPL, we are ready to move the code into an actual class that can be consumed by other projects. Go to Library1.fs in the same project and rename it to StockPriceRepositories.fs. Remove all code and replace it with this:

namespace NewCo.OptionsTradingProgram.Data

open System

open FSharp.Data

type csvProvider = CsvProvider<"http://ichart.finance.yahoo.com/table.csv?s=MSFT">

type YahooStockProvider() =

member this.GetData(stockSymbol: string) =

csvProvider.Load("http://ichart.finance.yahoo.com/table.csv?s=" + stockSymbol)

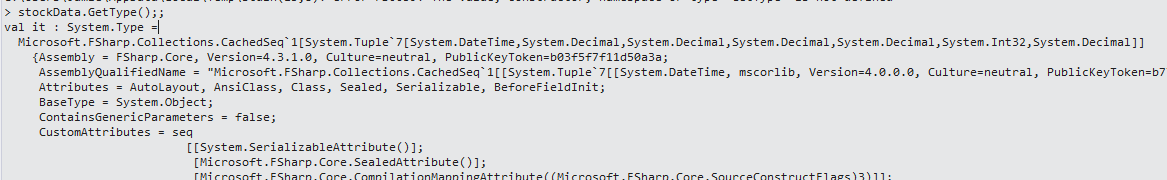
We now have a class that pulls data from Yahoo that we can use in our analysis. As we continue to progress with our project, we will want to pull this data down often. It would be much better to hold a sample of this data locally so that

1. Yahoo doesn’t throttle us
2. We have the same dataset every time for controlled experiments

We could hard-code in a sample dataset into memory, but that is a lot of typing. Instead, let’s pull the data down one time and store it on disk. That way, we can pull it up when needed. The barrier is that the type of the csvProvider is inferred. You can see this by entering this into the REPL:

stockData.GetType();;

Which returns:

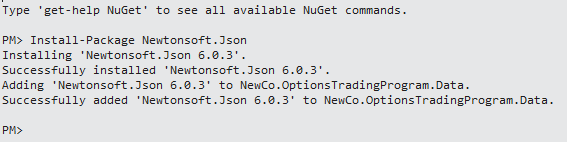


So once we start using different providers of the same type, we should define the type and then have a way of serializing and deserializing it. Let’s add Json.Net to our project so we can serialize and deserialize to Json.

Open the Package Manager Console and type this

Install-Package Newtonsoft.Json

You should get this



With Newtonsoft.Json installed, we can reference it in the REPL.

#r "C:\Users\Jamie\Desktop\NewCo.OptionsTradingProgram.Solution\packages\FSharp.Data.2.0.8\lib\portable-net40+sl5+wp8+win8\FSharp.Data.dll"

#r "C:\Users\Jamie\Desktop\NewCo.OptionsTradingProgram.Solution\packages\Newtonsoft.Json.6.0.3\lib\\net45\Newtonsoft.Json.dll"

open System

open FSharp.Data

open Newtonsoft.Json

Next, add this below Newtonsoft.Json

open System.IO

At the bottom of the script, add these lines:

type FileSystemStockProvider() =

member this.PutData(filePath:string, stockData) =

let serializedData = stockData

|> Seq.map(fun row -> JsonConvert.SerializeObject(row))

File.WriteAllLines(filePath,serializedData)

member this.GetData(filePath:string) =

let serializedData = File.ReadAllLines(filePath)

serializedData

|> Seq.map(fun row -> JsonConvert.DeserializeObject<(DateTime\*float\*float\*float\*float\*int\*float)>(row))

The entire script should look like this:

#r "C:\Users\Jamie\Desktop\NewCo.OptionsTradingProgram.Solution\packages\FSharp.Data.2.0.8\lib\portable-net40+sl5+wp8+win8\FSharp.Data.dll"

#r "C:\Users\Jamie\Desktop\NewCo.OptionsTradingProgram.Solution\packages\Newtonsoft.Json.6.0.3\lib\\net45\Newtonsoft.Json.dll"

open System

open FSharp.Data

open Newtonsoft.Json

open System.IO

type csvProvider = CsvProvider<"http://ichart.finance.yahoo.com/table.csv?s=MSFT">

type YahooStockProvider() =

member this.GetData(stockSymbol: string) =

csvProvider.Load("http://ichart.finance.yahoo.com/table.csv?s=" + stockSymbol).Rows

type FileSystemStockProvider() =

member this.PutData(filePath:string, stockData) =

let serializedData = stockData

|> Seq.map(fun row -> JsonConvert.SerializeObject(row))

File.WriteAllLines(filePath,serializedData)

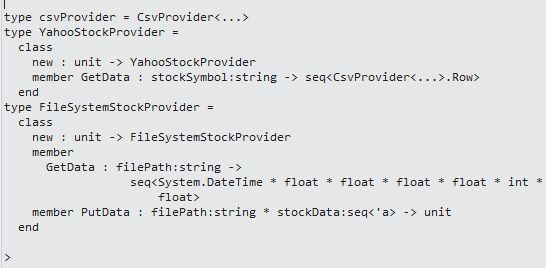
member this.GetData(filePath:string) =

let serializedData = File.ReadAllLines(filePath)

serializedData

|> Seq.map(fun row -> JsonConvert.DeserializeObject<(DateTime\*float\*float\*float\*float\*int\*float)>(row))

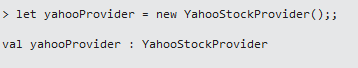
Highlight everything and hit ALT+ENTER. The REPL should look like:



In the REPL, enter this at the >

let yahooProvider = new YahooStockProvider();;

And you should get this



Next, enter this:

let yahooData = yahooProvider.GetData("MSFT");;

And you should get this:



Next, enter this:

let fileSystemProvider = new FileSystemStockProvider();;

And you should get this:



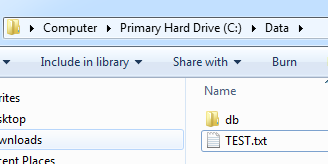
Next, enter this:

fileSystemProvider.PutData("C:\Data\TEST.txt",yahooData);;

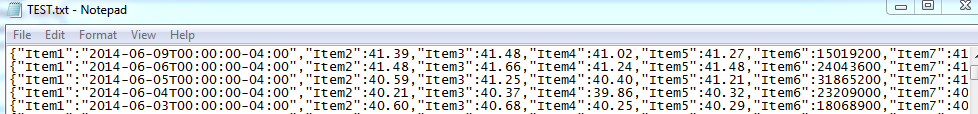
And you should get this:



And if you open your file system, you should see the file:



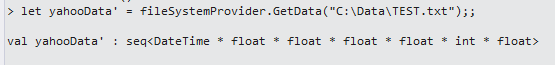
And if you open the file, you can see the contents:



Back to the REPL, enter this:

let yahooData' = fileSystemProvider.GetData("C:\Data\TEST.txt");;

And you should get this:



So now we have a way of getting data from Yahoo and read and writing it to the file system. Let’s take the code from the .fsx file and put it into the StockPriceRepositories file:

namespace NewCo.OptionsTradingProgram.Data

open System

open FSharp.Data

open Newtonsoft.Json

open System.IO

type csvProvider = CsvProvider<"http://ichart.finance.yahoo.com/table.csv?s=MSFT">

type YahooStockProvider() =

member this.GetData(stockSymbol: string) =

csvProvider.Load("http://ichart.finance.yahoo.com/table.csv?s=" + stockSymbol).Rows

type FileSystemStockProvider(filePath:string) =

member this.PutData(stockData) =

let serializedData = stockData

|> Seq.map(fun row -> JsonConvert.SerializeObject(row))

File.WriteAllLines(filePath,serializedData)

member this.GetData() =

let serializedData = File.ReadAllLines(filePath)

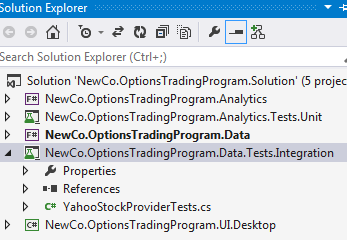
serializedData

|> Seq.map(fun row -> JsonConvert.DeserializeObject<(DateTime\*float\*float\*float\*float\*int\*float)>(row))

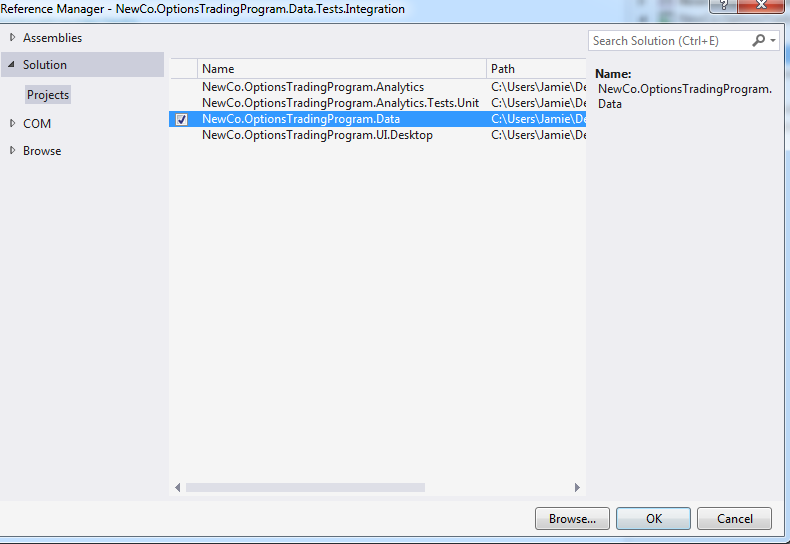
You will notice a subtle difference in the FileSystemStockProvider -> the filepath is now part of the constructor. This is so we can apply an interface to both the YahooStockProvider and the FileSystemStockProvider and swap them out at runtime, depending on our needs. Note that we are keeping the script file as part of the solution just to have. If we need to make additional changes to the code, we can always go into our script to “prove out” the concept and then bring the code into our module.

Next, let’s build a couple of integration tests so that other non-F# developers can see how to use our classes and so we have a repeatable way of verifying our code.

Open the newCo.OptionsTradingProgram.Data.Tests.Integration project and rename UnitTest1 to YahooStockProviderTests



Next, add a reference to the NewCo.OptionsTradingProgram.Data project from the test project:



Next, Updated the YahooStockProviderTests class to test a call to yahoo.

[TestClass]

public class YahooStockProviderTests

{

[TestMethod]

public void YahooStockProviderUsingMSFT\_ReturnsExpected()

{

YahooStockProvider stockProvider = new YahooStockProvider();

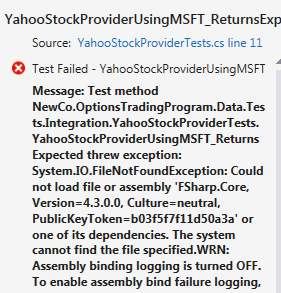
var data = stockProvider.GetData("MSFT");

Assert.IsNotNull(data);

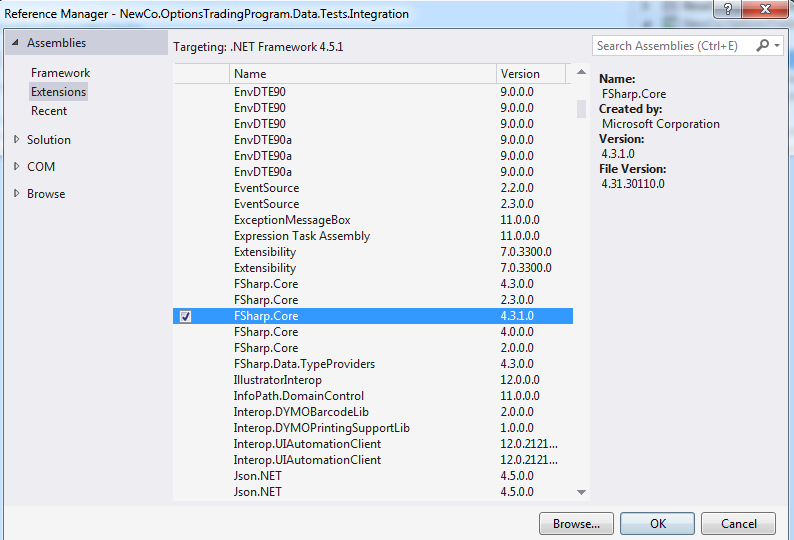
}

}

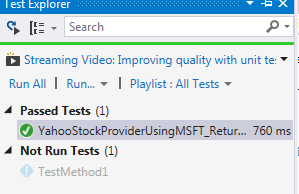
When you try and run it for the first time, it will fail with this message.



The problem is that FSharp has its own internal collection of types that are not part System.Core. The choice is to either add a reference to FSharp.core in the consuming application (like this C# test project) or to convert the FSharp types to a general .NET type. Since we are using type providers to allow inferred typing, we are going to do option #1. Open up the test project references (Assemblies->Extensions) and add a reference to FSharp.core (it might be 4.3.1 or 4.3.0 depending on your version of VS/Mono)



Now when you run the Test project, it now passes:



Add in another test class to the unit test project and call it FileSystemStockProviderTests. Add a couple of integration tests

[TestMethod]

public void PutData\_ReturnsExpected()

{

FileSystemStockProvider provider = new FileSystemStockProvider(@"C:\Data\TEST.txt");

var data = provider.GetData();

if (data != null)

{

provider.PutData(data);

}

}

[TestMethod]

public void GetData\_ReturnsExpected()

{

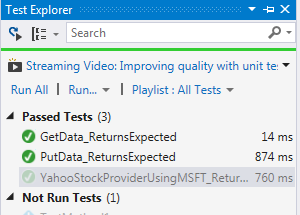
FileSystemStockProvider provider = new FileSystemStockProvider(@"C:\Data\TEST.txt");

var data = provider.GetData();

Assert.IsNotNull(data);

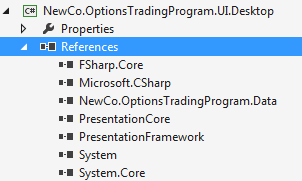
}

Run them to get green

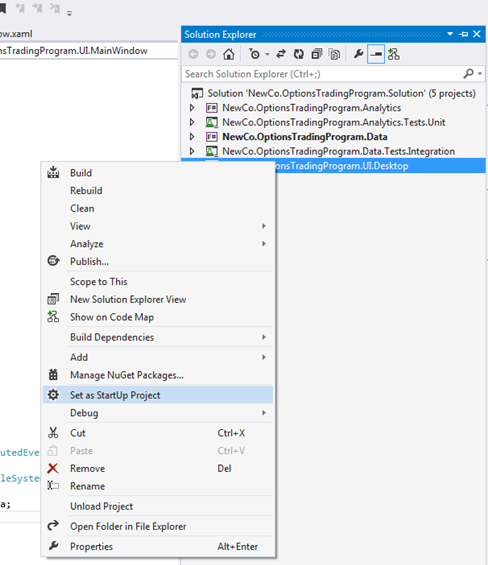


**Build a UI**

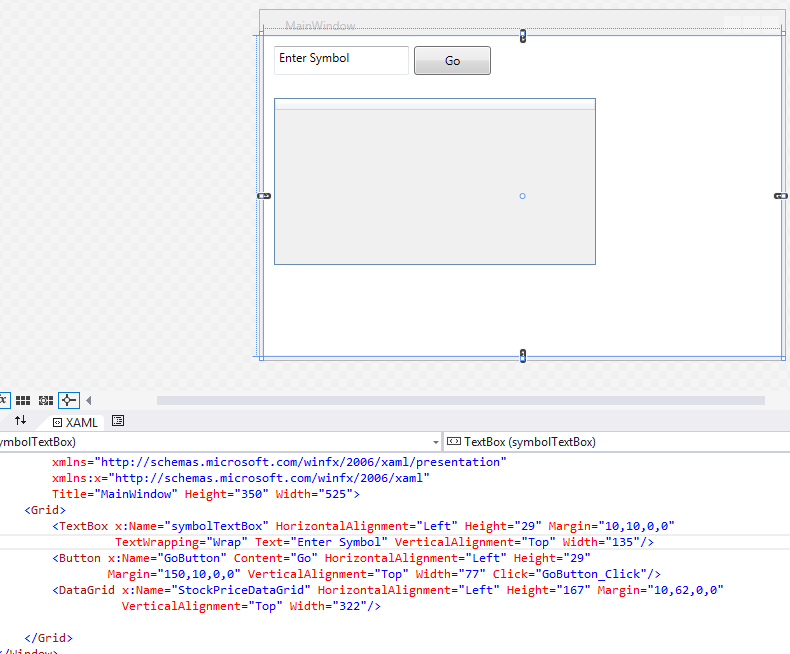
With the data layer now set, we are ready to build a UI that can consume our data/analytics. Open the NewCo.OptionsTradingProgram.UI.Desktop. Add a reference to NewCo.OptionsTradingProgram.Data and FSharp.Core



Next, set this project as the startup one:



In the Main Window designer, add a textbox to enter a symbol, a grid to display the data, and a button to get the data and put it in the grid

.

Go to the code behind and in the event handler for the button, add the following code:

private void GoButton\_Click(object sender, RoutedEventArgs e)

{

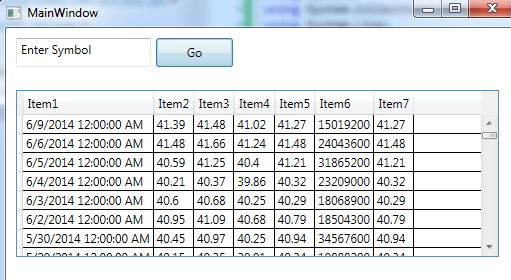
FileSystemStockProvider provider = new FileSystemStockProvider(@"C:\Data\TEST.txt");

var data = provider.GetData();

this.StockPriceDataGrid.ItemsSource = data;

}

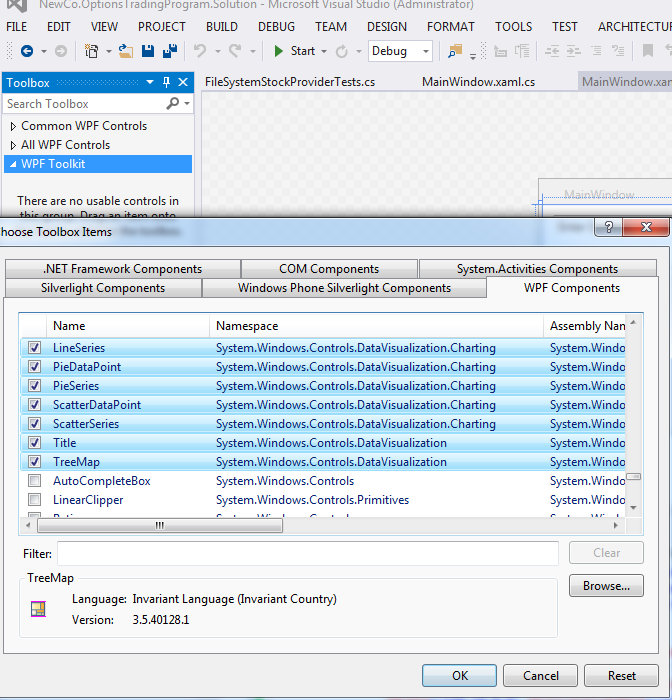
Run the project and you should get something like:



Note that you don’t need to enter the symbol b/c we are pulling static data from disk.

Next, install the WPF Toolkit from codeplex: <http://wpf.codeplex.com/releases/view/40535>

Back in the designer, open the toolbox and add a tab called WPF Toolkit. Right click and add all controls in the System.Windows.Controls.DataVisualization.Toolkit namespace. You can sort by assembly name in the dialog box



Back in the page designer, drag a bar series from the toolbox onto the canvas. Call it stockPriceLineGraph

<Grid Margin="0,0,0,-261">

<Grid.RowDefinitions>

<RowDefinition Height="77\*"/>

<RowDefinition Height="17\*"/>

</Grid.RowDefinitions>

<TextBox x:Name="symbolTextBox" HorizontalAlignment="Left" Height="29" Margin="10,10,0,0"

TextWrapping="Wrap" Text="Enter Symbol" VerticalAlignment="Top" Width="135"/>

<Button x:Name="GoButton" Content="Go" HorizontalAlignment="Left" Height="29"

Margin="150,10,0,0" VerticalAlignment="Top" Width="77" Click="GoButton\_Click"/>

<DataGrid x:Name="StockPriceDataGrid" HorizontalAlignment="Left" Height="167" Margin="10,62,0,0"

VerticalAlignment="Top" Width="483"/>

<chartingToolkit:Chart x:Name="chart" Width="350" Height="250">

<chartingToolkit:Chart.Series>

<chartingToolkit:LineSeries x:Name="stockPriceLineGraph"

IsSelectionEnabled="True"

HorizontalAlignment="Left" Height="178"

Foreground="Black" VerticalAlignment="Top" Width="220"/>

</chartingToolkit:Chart.Series>

</chartingToolkit:Chart>

</Grid>

In the code behind, take the top 20 rows from the provider and add those points to the graph:

private void GoButton\_Click(object sender, RoutedEventArgs e)

{

FileSystemStockProvider provider = new FileSystemStockProvider(@"C:\Data\TEST.txt");

var stockPrices = provider.GetData().Take(10);

this.StockPriceDataGrid.ItemsSource = stockPrices;

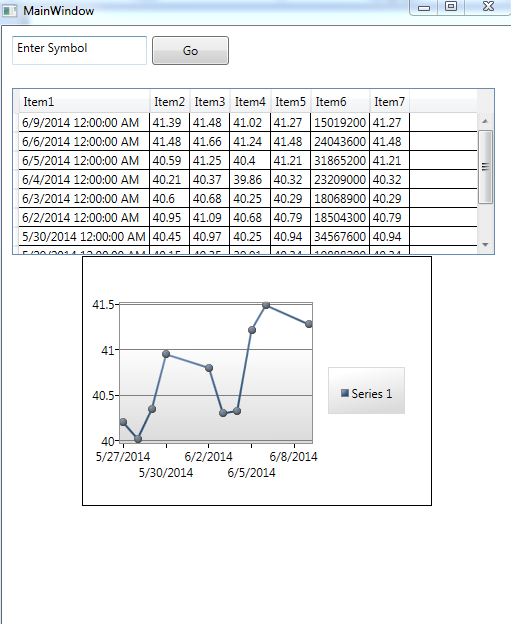
this.stockPriceLineGraph.DependentValuePath="Item7";

this.stockPriceLineGraph.IndependentValuePath = "Item1";

this.stockPriceLineGraph.ItemsSource = stockPrices;

}

And you should see:



Back in the page designer, add another series to the graph

<chartingToolkit:Chart x:Name="chart" Width="350" Height="250">

<chartingToolkit:Chart.Series>

<chartingToolkit:LineSeries x:Name="stockPriceLineGraph"

IsSelectionEnabled="True"

HorizontalAlignment="Left" Height="178"

Foreground="Black" VerticalAlignment="Top" Width="220"/>

<chartingToolkit:LineSeries x:Name="stockPriceLineGraph2"

IsSelectionEnabled="True"

HorizontalAlignment="Left" Height="178"

Foreground="Red" VerticalAlignment="Top" Width="220"/>

</chartingToolkit:Chart.Series>

In the code behind, bind the second series to data element #4

private void GoButton\_Click(object sender, RoutedEventArgs e)

{

FileSystemStockProvider provider = new FileSystemStockProvider(@"C:\Data\TEST.txt");

var stockPrices = provider.GetData().Take(10);

this.StockPriceDataGrid.ItemsSource = stockPrices;

this.stockPriceLineGraph.DependentValuePath="Item7";

this.stockPriceLineGraph.IndependentValuePath = "Item1";

this.stockPriceLineGraph.ItemsSource = stockPrices;

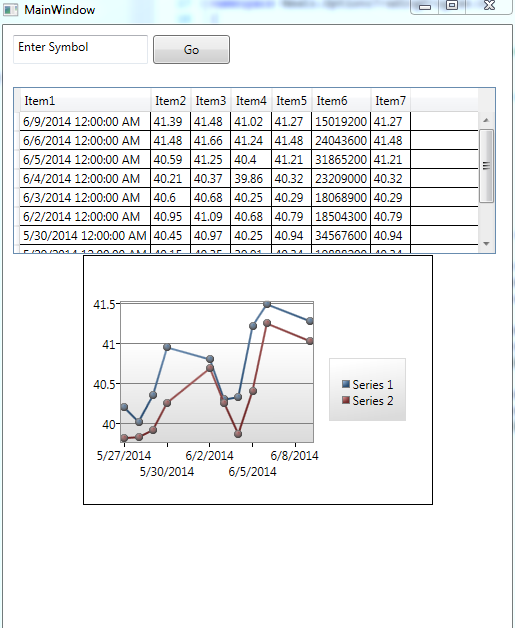
this.stockPriceLineGraph2.DependentValuePath = "Item4";

this.stockPriceLineGraph2.IndependentValuePath = "Item1";

this.stockPriceLineGraph2.ItemsSource = stockPrices;

}

And run again



**Implement Basic Statistics**

Open NewCo.OptionsTradingProgram.Analytics. Open the Script.fsx file and remove everything in it. There are some mathematical formulas we will need to use for our analysis. This is an area where F# really shines because it makes working with mathematical formulas very natural. In the script file, create an array of numbers 1 to 6 for analysis

let testData = [1.0 .. 6.0]

Hit ALT+ENTER and you can see the results in the REPL



In the script file, sum the numbers in the array

let testData = [1.0 .. 6.0]

let sum1 = Seq.sum testData



Note that you can sum the numbers using a ‘pipe forward’ syntax. This is the syntax that I will be using by default from now on.

let testData = [1.0 .. 6.0]

let sum1 = Seq.sum testData

let sum2 = testData |> Seq.sum



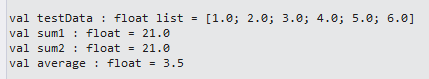
In the script file, average the numbers in the array

let testData = [1.0 .. 6.0]

let sum1 = Seq.sum testData

let sum2 = testData |> Seq.sum

let average = testData |> Seq.average



In the script file, find the minimum value the numbers in the array

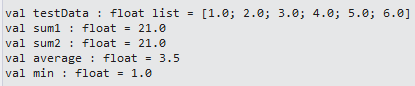
let testData = [1.0 .. 6.0]

let sum1 = Seq.sum testData

let sum2 = testData |> Seq.sum

let average = testData |> Seq.average

let min = testData |> Seq.min



In the script file, find the maximum value the numbers in the array

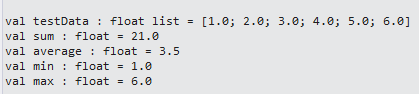
let testData = [1.0 .. 6.0]

let sum = testData |> Seq.sum

let average = testData |> Seq.average

let min = testData |> Seq.min

let max = testData |> Seq.max



In the script file, find all even numbers in the array

let testData = [1.0 .. 6.0]

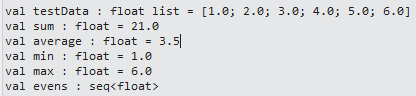
let sum = testData |> Seq.sum

let average = testData |> Seq.average

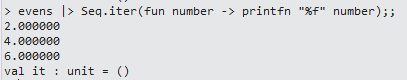
let min = testData |> Seq.min

let max = testData |> Seq.max

let evens = testData |> Seq.filter(fun number -> number % 2. = 0.)



Note that the even numbers are in their own list. To actually see each member, go to the REPL and type this



In the script file, add 1 number to every number in the array

let testData = [1.0 .. 6.0]

let sum = testData |> Seq.sum

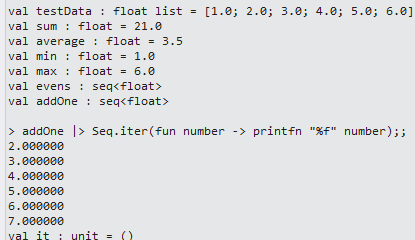
let average = testData |> Seq.average

let min = testData |> Seq.min

let max = testData |> Seq.max

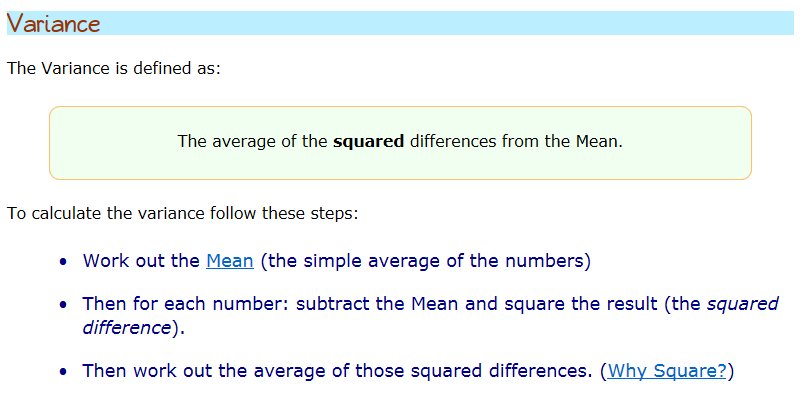
let evens = testData |> Seq.filter(fun number -> number % 2. = 0.)

let addOne = testData |> Seq.map(fun number -> number + 1.)



Now let’s create the formula for variance. If you are not familiar with variance, here is a good description: http://www.mathsisfun.com/data/standard-deviation.html

And here is the methodology that the page uses



In the script file, create the variance. Notice how there is a one to one correspondence between F# and to the written steps:

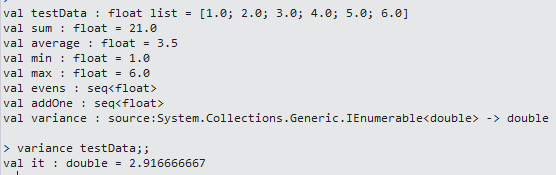
open System.Collections.Generic

let variance (source:IEnumerable<double>) =

let mean = Seq.average source

let deltas = Seq.map(fun x -> pown(x-mean) 2) source

Seq.average deltas



Next, let’s calculate the standard deviation. You can refer back to that Math Is Fun page for a full explanation of standard deviation. The formula is:



In our code, add in the standardDeviation:

open System.Collections.Generic

let testData = [1.0 .. 6.0]

let sum = testData |> Seq.sum

let average = testData |> Seq.average

let min = testData |> Seq.min

let max = testData |> Seq.max

let evens = testData |> Seq.filter(fun number -> number % 2. = 0.)

let addOne = testData |> Seq.map(fun number -> number + 1.)

//http://www.mathsisfun.com/data/standard-deviation.html

let variance (source:IEnumerable<double>) =

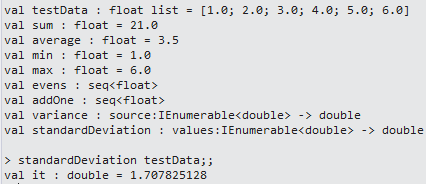
let mean = Seq.average source

let deltas = Seq.map(fun x -> pown(x-mean) 2) source

Seq.average deltas

let standardDeviation(values:IEnumerable<double>) =

sqrt(variance(values))



Next, let’s add in a moving average. If you are not familiar with moving average, here is a good explanation: <http://www.investopedia.com/terms/m/movingaverage.asp>

To calculate the moving average, follow these steps:

* Break a list of numbers into a series of smaller lists
* Calculate the average on those sublists

Again, there is a one to one correspondence between F# and how you would describe the calculation in English:

let movingAverage(values:IEnumerable<double>, windowSize:int)=

values

|> Seq.windowed (windowSize)

|> Seq.map(fun window -> window |> Seq.average)



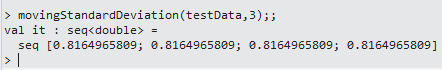
Next, let’s add a moving standard deviation. It is the same concept as a moving average and the code is identical except for 1 place:

let movingStandardDeviation(values:IEnumerable<double>, windowSize:int)=

values

|> Seq.windowed (windowSize)

|> Seq.map(fun window -> window |> standardDeviation)



Next, let’s add a Bollinger bands. This is the first financial-related formula we have. If you are not familiar with Bollinger bands, a good explanation is here: <http://www.investopedia.com/terms/b/bollingerbands.asp>. To calculate Bollinger bands, you need to get 2 standard deviations away from a simple moving average:

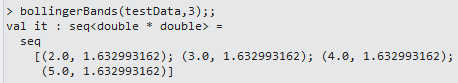
let bollingerBands (values:IEnumerable<double>, windowSize:int)=

let movingAverage = movingAverage(values,windowSize)

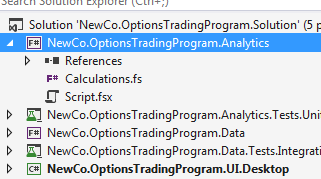
let movingStandardDeviation = movingStandardDeviation(values,windowSize)

let movingStandardDeviation' = movingStandardDeviation |> Seq.map(fun window -> window \* 2.)

Seq.zip movingAverage movingStandardDeviation'



Let’s take a break from our scripting and make a chart from our Bollinger bands. In the solution explorer, rename Library1.fs to Calculations.fs



Open Calculations.fs and rename Class1() to Calculations()

namespace NewCo.OptionsTradingProgram.Analytics

type Calculations() =

member this.X = "F#"

Next, add in the reference to System.Collections.Generic and the variance calculation

namespace NewCo.OptionsTradingProgram.Analytics

open System.Collections.Generic

type Calculations() =

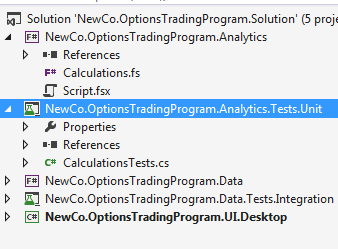
member this.Variance (source:IEnumerable<double>) =

let mean = Seq.average source

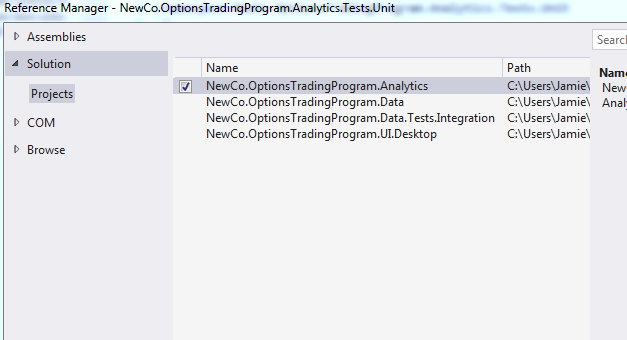
let deltas = Seq.map(fun x -> pown(x-mean) 2) source

Seq.average deltas

Next, in the solution explorer, rename UnitTest1 to CalculationsTests



Open NewCo.OptionsTradingProgram.Analytics.Tests.Unit and add a reference to NewCo.OptionsTradingProgram.Analytics



Open CalculationsTests, rename TestMethod1 to CalculateVarianceUsingValidData\_ReturnsExpected and implement it like so

using System;

using Microsoft.VisualStudio.TestTools.UnitTesting;

using NewCo.OptionsTradingProgram.Analytics;

namespace NewCo.OptionsTradingProgram.Analytics.Tests.Unit

{

[TestClass]

public class CalculationsTests

{

[TestMethod]

public void CalculateVarianceUsingValidData\_ReturnsExpected()

{

var testData = new Double[6] { 1, 2, 3, 4, 5, 6 };

var calculations = new Calculations();

var variance = calculations.Variance(testData);

var expected = 2.916666667;

var actual = Math.Round(variance, 9);

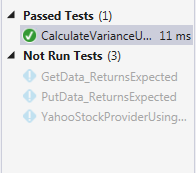
Assert.AreEqual(expected, actual);

}

}

}

Run your unit test and get green:



Go back to the Calculations.fs file and add in the remaining formulas from your script file. You will replace “let” with “member this.” and apply Pascal casing to the function names:

namespace NewCo.OptionsTradingProgram.Analytics

open System.Collections.Generic

type Calculations() =

member this.Variance (source:IEnumerable<double>) =

let mean = Seq.average source

let deltas = Seq.map(fun x -> pown(x-mean) 2) source

Seq.average deltas

member this.StandardDeviation(values:IEnumerable<double>) =

sqrt(this.Variance(values))

member this.MovingAverage(values:IEnumerable<double>, windowSize:int)=

values

|> Seq.windowed (windowSize)

|> Seq.map(fun window -> window |> Seq.average)

member this.MovingStandardDeviation(values:IEnumerable<double>, windowSize:int)=

values

|> Seq.windowed (windowSize)

|> Seq.map(fun window -> window |> this.StandardDeviation)

member this.BollingerBands (values:IEnumerable<double>, windowSize:int)=

let movingAverage = this.MovingAverage(values,windowSize)

let movingStandardDeviation = this.MovingStandardDeviation(values,windowSize)

let movingStandardDeviation' = movingStandardDeviation |> Seq.map(fun window -> window \* 2.)

Seq.zip movingAverage movingStandardDeviation'

Compile, go to your test project, and add in the following tests:

[TestMethod]

public void CalcualteStandardDeviationUsingValidData\_ReturnsExpected()

{

var testData = new Double[6] { 1, 2, 3, 4, 5, 6 };

var calculations = new Calculations();

var standadDeviation = calculations.StandardDeviation(testData);

var expected = 1.707825128;

var actual = Math.Round(standadDeviation,9);

Assert.AreEqual(expected, actual);

}

[TestMethod]

public void CalculateMovingAverageUsingValidData\_ReturnsExpected()

{

var testData = new Double[6] { 1, 2, 3, 4, 5, 6 };

var calculations = new Calculations();

var movingAverage = calculations.MovingAverage(testData,3);

var expected = 4;

var actual = new List<Double>(movingAverage).Count;

Assert.AreEqual(expected, actual);

}

[TestMethod]

public void CalculateMovingStandardDeviationUsingValidData\_ReturnsExpected()

{

var testData = new Double[6] { 1, 2, 3, 4, 5, 6 };

var calculations = new Calculations();

var movingStandardDeviation = calculations.MovingStandardDeviation(testData,3);

var expected = 4;

var actual = new List<Double>(movingStandardDeviation).Count;

Assert.AreEqual(expected, actual);

}

[TestMethod]

public void CalculateBollingerBandsUsingValidData\_ReturnsExpected()

{

var testData = new Double[6] { 1, 2, 3, 4, 5, 6 };

var calculations = new Calculations();

var bollingerBands = calculations.BollingerBands(testData,3);

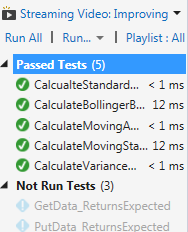
var expected = 4;

var actual = new List<Tuple<Double,Double>>(bollingerBands).Count;

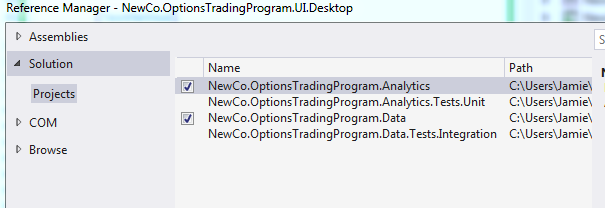
Assert.AreEqual(expected, actual);

}

Note you will have to add a reference to System.Collections.Generics. Once the tests are ready, run them and get green



Finally, open NewCo.OptionsTrandingProgram.UI.Desktop and add a reference to NewCo.OptionsTradingProgram.Analytics.



Open the MainWindow.xaml file and alter the chart to have three line series:

<chartingToolkit:Chart x:Name="chart" Width="350" Height="250">

<chartingToolkit:Chart.Series>

<chartingToolkit:LineSeries x:Name="stockPriceLineGraph"

IsSelectionEnabled="True"

HorizontalAlignment="Left" Height="178"

Foreground="Black" VerticalAlignment="Top" Width="220"/>

<chartingToolkit:LineSeries x:Name="stockPriceLineGraph2"

IsSelectionEnabled="True"

HorizontalAlignment="Left" Height="178"

Foreground="Red" VerticalAlignment="Top" Width="220"/>

<chartingToolkit:LineSeries x:Name="stockPriceLineGraph3"

IsSelectionEnabled="True"

HorizontalAlignment="Left" Height="178"

Foreground="Green" VerticalAlignment="Top" Width="220"/>

</chartingToolkit:Chart.Series>

</chartingToolkit:Chart>

Open the MainWindow.xaml.cs file. Add a reference to NewCo.OptionsTradingProgram.Analytics. Next, alter the GoButton\_Click event handler like this:

private void GoButton\_Click(object sender, RoutedEventArgs e)

{

FileSystemStockProvider provider = new FileSystemStockProvider(@"C:\Data\TEST.txt");

var stockPrices = provider.GetData().Take(20);

this.StockPriceDataGrid.ItemsSource = stockPrices;

var adjustedClosePrices = from stockPrice in stockPrices

select stockPrice.Item7;

var dates = from stockPrice in stockPrices.Skip(2)

select new { stockPrice.Item1 };

var calculations = new Calculations();

var movingAverage = calculations.MovingAverage(adjustedClosePrices, 3);

var movingAverages = dates.Zip(movingAverage, (d, p) => new { date=d.Item1, price=p});

var bollingerBands = calculations.BollingerBands(adjustedClosePrices, 3);

var upperBandBands = dates.Zip(bollingerBands, (d, bb) => new { date = d.Item1, upperBand = bb.Item1 + (bb.Item2 \* 2) });

var lowerBandBands = dates.Zip(bollingerBands, (d, bb) => new { date = d.Item1, lowerBand = bb.Item1 + (bb.Item2 \* 2) \* -1 });

this.stockPriceLineGraph.DependentValuePath = "price";

this.stockPriceLineGraph.IndependentValuePath = "date";

this.stockPriceLineGraph.ItemsSource = movingAverages;

this.stockPriceLineGraph2.DependentValuePath = "upperBand";

this.stockPriceLineGraph2.IndependentValuePath = "date";

this.stockPriceLineGraph2.ItemsSource = upperBandBands;

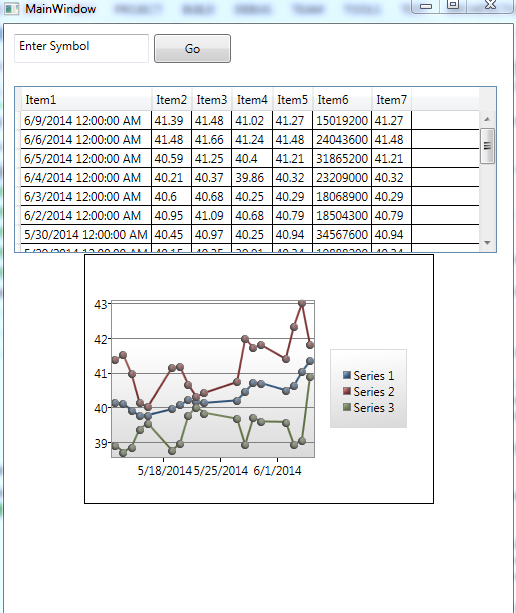
this.stockPriceLineGraph3.DependentValuePath = "lowerBand";

this.stockPriceLineGraph3.IndependentValuePath = "date";

this.stockPriceLineGraph3.ItemsSource = lowerBandBands;

}

You will notice we are using Linq, which has almost identical syntax to F#. Indeed, if you understand F#, Linq because very accessible. Unfortunately, the reverse is not true. In any event, if you run the application, you will get something like this:



**Implement Non-Basic Statistics**

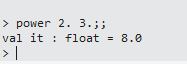
With these basic statistics under our belt, it is time to try our hand at some more advanced statistics. Open NewCo.OptionsTradingProgram.Analytics and open the Script.fsx file. Below the bollingerBands function, create a function to calculate the power of a number.

let power baseNumber exponent = exp(exponent \* log(baseNumber))

and you should get this:



In the REPL, enter in this



And you can see two cubed calculated.

A quick side note about this calculation.

* First, this is how the book implements the function:

let pow x n = exp(n\*log(x))

which I think that is makes the code much harder to read than it has to be

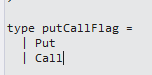
* Second, there is already a power function in the FSharp language spec. It is called pown. It was used in the variance calculation that you have already seen.

With power out of the way, let’s build a flag to determine if we are calculating a put or call.

Enter this into the script:

type putCallFlag = Put | Call

and you should get this in the REPL:



Next, let’s create a cumulative distribution function. If you are not familiar with the cumulative distribution function, you can read about it here: <http://en.wikipedia.org/wiki/Cumulative_distribution_function>. In the script file, enter this

let cumulativeDistribution (x) =

let a1 = 0.31938153

let a2 = -0.356563782

let a3 = 1.781477937

let a4 = -1.821255978

let a5 = 1.330274429

let pi = 3.141592654

let l = abs(x)

let k = 1.0 / (1.0 + 0.2316419 \* l)

let a1' = a1\*k

let a2' = a2\*k\*k

let a3' = a3\*(power k 3.0)

let a4' = a4\*(power k 4.0)

let a5' = a5\*(power k 5.0)

let w1 = 1.0/sqrt(2.0\*pi)

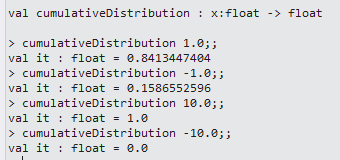
let w2 = exp(-l\*l/2.0)

let w3 = a1'+a2'+a3'+a4'+a5'

let w = 1.0-w1\*w2\*w3

if x < 0.0 then 1.0 - w else w

After hitting ALT+ENTER, you can enter some values in the REPL to see the results:



Note that this code is different than the book as I broke down his long formula into smaller subformulas.

With the supporting functions out of the way, let’s implement the Black-Scholes formula.

First, create a data structure to pass in the needed inputs to the function

type blackScholesInputData =

{stockPrice:float;

strikePrice:float;

timeToExpiry:float;

interestRate:float;

volatility:float}

Next, create the actual Black-Scholes calculation:

let blackScholes(inputData:blackScholesInputData, putCallFlag:putCallFlag)=

let sx = log(inputData.stockPrice / inputData.strikePrice)

let rv = inputData.interestRate+inputData.volatility\*inputData.volatility\*0.5

let rvt = rv\*inputData.timeToExpiry

let vt = (inputData.volatility\*sqrt(inputData.timeToExpiry))

let d1=(sx + rvt)/vt

let d2=d1-vt

match putCallFlag with

| Put ->

let xrt = inputData.strikePrice\*exp(-inputData.interestRate\*inputData.timeToExpiry)

let cdD1 = xrt\*cumulativeDistribution(-d2)

let cdD2 = inputData.stockPrice\*cumulativeDistribution(-d1)

cdD1-cdD2

| Call ->

let xrt = inputData.strikePrice\*exp(-inputData.interestRate\*inputData.timeToExpiry)

let cdD1 = inputData.stockPrice\*cumulativeDistribution(d1)

let cdD2 = xrt\*cumulativeDistribution(d2)

cdD1-cdD2

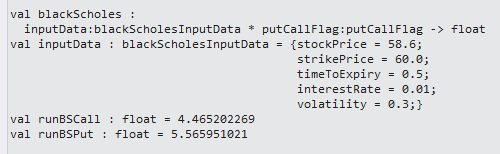
Also, enter in some test data:

let inputData = {stockPrice=58.60;strikePrice=60.;timeToExpiry=0.5;interestRate=0.01;volatility=0.3}

let runBSCall = blackScholes(inputData,Call)

let runBSPut = blackScholes(inputData,Put)

Hitting ALT+ENTER gives this in the REPL:



Note that the Time To Maturity is expressed in years right now. Add in a helper function to the script file

let daysToYears day = (float day) / 365.25

let ThirityDayEquiv = daysToYears 30

Hitting ALT+ENTER gives this in the REPL:



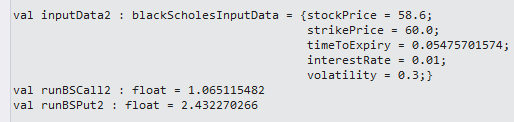
So the preceding example using a 20 day maturity:

let inputData2 = {stockPrice=58.60;strikePrice=60.;timeToExpiry=(daysToYears 20);interestRate=0.01;volatility=0.3}

let runBSCall2 = blackScholes(inputData2,Call)

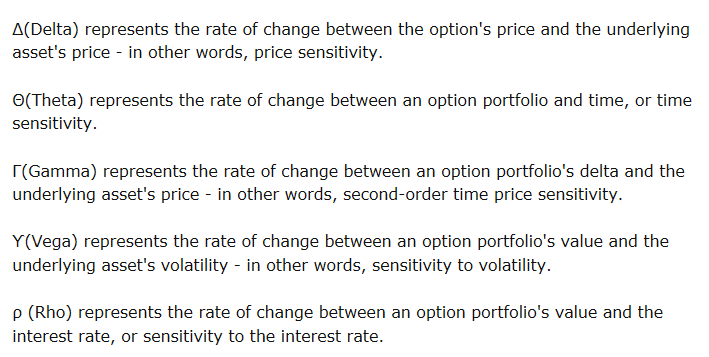
let runBSPut2 = blackScholes(inputData2,Put)

Gives



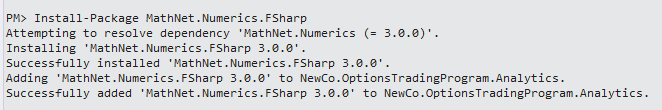
With the Black Scholes under our belt, let’s add a few more statistics that use the Black-Scholes formula in their calculations. They are commonly referred to as “the greeks” and include the following formulas: Delta, Vega, Theta, Rho, and Gamma. Here is a good explanation of them: <http://www.investopedia.com/terms/g/greeks.asp>

And in chart form:



Before we start implementing the Greeks, we need to add in 1 more helper method. We need to have a normal distribution created for some of the greeks. We could hand-write it, but there is a great library called MathNet.Numerics that has a pre-created function for this very purpose.

Go to the NuGet Package Manager Console and enter in the following command:



In your script file, enter in the following code:

#r @"E:\Documents\TriNug\FSharpLab\_20140615\NewCo.OptionsTradingProgram.Solution\packages\MathNet.Numerics.3.0.0\lib\net40\MathNet.Numerics.dll"

#r @"E:\Documents\TriNug\FSharpLab\_20140615\NewCo.OptionsTradingProgram.Solution\packages\MathNet.Numerics.FSharp.3.0.0\lib\net40\MathNet.Numerics.FSharp.dll"

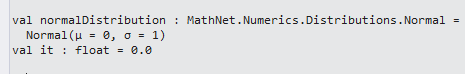
open MathNet.Numerics.Distributions;

let normalDistribution = new Normal(0.0, 1.0)

normalDistribution.Density(100.0)

(the physical path will be different on your machine)

Hit ALT+ENTER and get:



Delta

let blackScholesDelta (inputData:blackScholesInputData, putCallFlag:putCallFlag) =

let sx = log(inputData.stockPrice / inputData.strikePrice)

let rv = inputData.interestRate+inputData.volatility\*inputData.volatility\*0.5

let rvt = rv\*inputData.timeToExpiry

let vt = (inputData.volatility\*sqrt(inputData.timeToExpiry))

let d1=(sx + rvt)/vt

match putCallFlag with

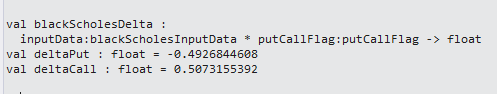
| Put -> cumulativeDistribution(d1) - 1.0

| Call -> cumulativeDistribution(d1)

let deltaPut = blackScholesDelta(inputData, Put)

let deltaCall = blackScholesDelta(inputData, Call)

Assuming you have not altered the inputData, you will get:



Gamma

let blackScholesGamma (inputData:blackScholesInputData) =

let sx = log(inputData.stockPrice / inputData.strikePrice)

let rv = inputData.interestRate+inputData.volatility\*inputData.volatility\*0.5

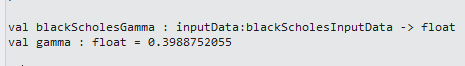
let rvt = rv\*inputData.timeToExpiry

let vt = (inputData.volatility\*sqrt(inputData.timeToExpiry))

let d1=(sx + rvt)/vt

normalDistribution.Density(d1)

let gamma = blackScholesGamma(inputData)



Vega

let blackScholesVega (inputData:blackScholesInputData) =

let sx = log(inputData.stockPrice / inputData.strikePrice)

let rv = inputData.interestRate+inputData.volatility\*inputData.volatility\*0.5

let rvt = rv\*inputData.timeToExpiry

let vt = (inputData.volatility\*sqrt(inputData.timeToExpiry))

let d1=(sx + rvt)/vt

inputData.stockPrice\*normalDistribution.Density(d1)\*sqrt(inputData.timeToExpiry)

let vega = blackScholesVega(inputData)



Theta

let blackScholesTheta (inputData:blackScholesInputData, putCallFlag:putCallFlag) =

let sx = log(inputData.stockPrice / inputData.strikePrice)

let rv = inputData.interestRate+inputData.volatility\*inputData.volatility\*0.5

let rvt = rv\*inputData.timeToExpiry

let vt = (inputData.volatility\*sqrt(inputData.timeToExpiry))

let d1=(sx + rvt)/vt

let d2=d1-vt

match putCallFlag with

| Put ->

let ndD1 = inputData.stockPrice\*normalDistribution.Density(d1)\*inputData.volatility

let ndD1' = ndD1/(2.0\*sqrt(inputData.timeToExpiry))

let rx = inputData.interestRate\*inputData.strikePrice

let rt = exp(-inputData.interestRate\*inputData.timeToExpiry)

let cdD2 = rx\*rt\*cumulativeDistribution(-d2)

-(ndD1')+cdD2

| Call ->

let ndD1 = inputData.stockPrice\*normalDistribution.Density(d1)\*inputData.volatility

let ndD1' = ndD1/(2.0\*sqrt(inputData.timeToExpiry))

let rx = inputData.interestRate\*inputData.strikePrice

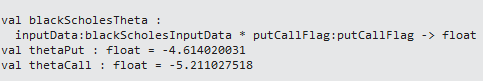
let rt = exp(-inputData.interestRate\*inputData.timeToExpiry)

let cdD2 = cumulativeDistribution(d2)

-(ndD1')-rx\*rt\*cdD2

let thetaPut = blackScholesTheta(inputData, Put)

let thetaCall = blackScholesTheta(inputData, Call)



Rho

let blackScholesRho (inputData:blackScholesInputData, putCallFlag:putCallFlag) =

let sx = log(inputData.stockPrice / inputData.strikePrice)

let rv = inputData.interestRate+inputData.volatility\*inputData.volatility\*0.5

let rvt = rv\*inputData.timeToExpiry

let vt = (inputData.volatility\*sqrt(inputData.timeToExpiry))

let d1=(sx + rvt)/vt

let d2=d1-vt

match putCallFlag with

| Put ->

let xt = inputData.strikePrice\*inputData.timeToExpiry

let rt = exp(-inputData.interestRate\*inputData.timeToExpiry)

-xt\*rt\*cumulativeDistribution(-d2)

| Call ->

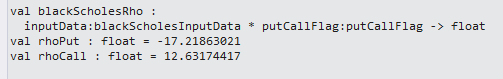
let xt = inputData.strikePrice\*inputData.timeToExpiry

let rt = exp(-inputData.interestRate\*inputData.timeToExpiry)

xt\*rt\*cumulativeDistribution(d2)

let rhoPut = blackScholesRho(inputData, Put)

let rhoCall = blackScholesRho(inputData, Call)



With the Greeks out of the way, we have 1 more function we need to write -> the Monte Carlo method. If you are not familiar with the Monte Carlo method, here is a good explanation. We will use it to produce a a number that we can compare the Black-Scholes number to. The difference is the XXX.

Open up the script file and enter this:

type monteCarloInputData =

{stockPrice:float;

strikePrice:float;

timeToExpiry:float;

interestRate:float;

volatility:float}

let priceAtMaturity (inputData:monteCarloInputData, randomValue:float) =

let s = inputData.stockPrice

let rv = (inputData.interestRate-inputData.volatility\*inputData.volatility/2.0)

let rvt = rv\*inputData.timeToExpiry

let vr = inputData.volatility\*randomValue

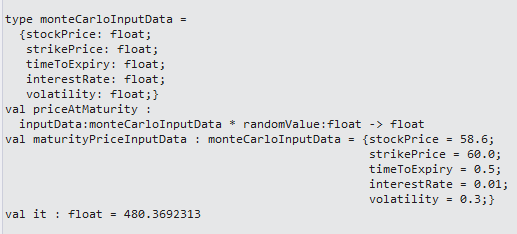
let t = sqrt(inputData.timeToExpiry)

s\*exp(rvt+vr\*t)

let maturityPriceInputData = {stockPrice=58.60;strikePrice=60.0;timeToExpiry=0.5;interestRate=0.01;volatility=0.3}

priceAtMaturity(maturityPriceInputData, 10.0)

You will get this from the REPL



Back in the script file:

let monteCarlo(inputData: monteCarloInputData, randomValues:seq<float>) =

randomValues

|> Seq.map(fun randomValue -> priceAtMaturity(inputData,randomValue) - inputData.strikePrice )

|> Seq.average

let random = new System.Random()

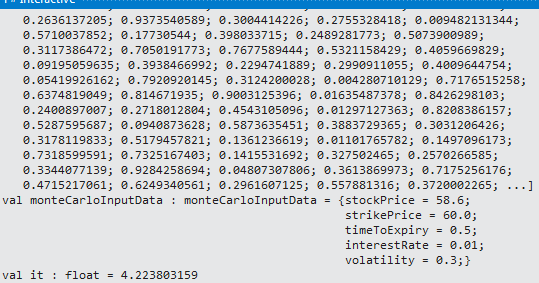
let rnd() = random.NextDouble()

let data = [for i in 1 .. 1000 -> rnd() \* 1.0]

let monteCarloInputData = {stockPrice=58.60;strikePrice=60.0;timeToExpiry=0.5;interestRate=0.01;volatility=0.3;}

monteCarlo(monteCarloInputData,data)

You will get something like this in the REPL:



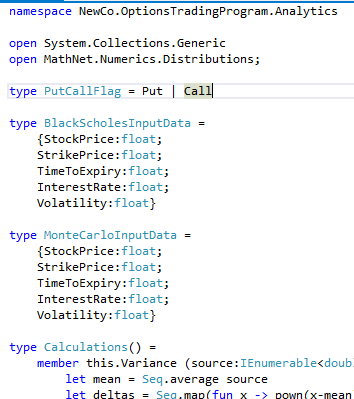
Congratulations. You have finished coding the advanced statistics. Now, let’s put it into our application

Open up the Calculations.fs file and copy/paste in the references at the top of the file.

* MathNet.Numerics.Distributions

Next, copy/paste in the types near the top of the module:

* the putCallFlag
* the blackScholesInputData
* the monteCarloInputData



Note you will change the casing of the types because these types will be consumed from non-F# projects.

Next, copy/pate in the functions into the module:

* + the power function
  + the cumulativeDistribution function,
  + the blackScholes function
  + the daysToYears function
  + the greek functions
  + the priceAtMaturity function
  + the monteCarlo function

You will need to change ‘let’ to ‘member this.’ Next, you will need to change the casing of your types (‘StockPrice” not ‘stockPrice’). Also, there are a couple of places where a function refers to another function. Prepend ‘this.’ Will allow the code to compile. Finally, you will need to move the setting of the NormalDistribution’s density into the greek functions that use it (Gamma and Vega). For example, Gaamma looks like this now:

member this.BlackScholesGamma (inputData:BlackScholesInputData) =

let sx = log(inputData.StockPrice / inputData.StrikePrice)

let rv = inputData.InterestRate+inputData.Volatility\*inputData.Volatility\*0.5

let rvt = rv\*inputData.TimeToExpiry

let vt = (inputData.Volatility\*sqrt(inputData.TimeToExpiry))

let d1=(sx + rvt)/vt

this.NormalDistribution.Density(100.0) |> ignore

this.NormalDistribution.Density(d1)

Once you have the module compiling, the next step is…. Unit tests!

Open the NewCo.OptionsTradingProgram.Analytics,Tests.Unit project. Open the the Calculations Tests and enter in unit tests for each of the functions that you crated. Use the test data from the script file to ensure that you have moved things correctly. For example, here is the power unit test:

[TestMethod]

public void PowerUsingValidData\_ReturnsExpected()

{

var calculations = new Calculations();

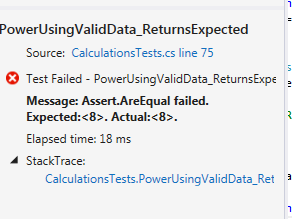
Double expected = 8;

Double actual = Math.Round(calculations.Power(2.0, 3.0),0);

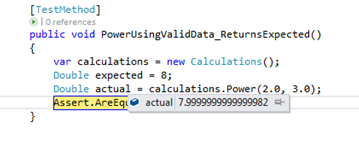
Assert.AreEqual(expected, actual);

}

An amusing side-note, take off the math.round on the actual calculation and run your test. You will get this:



Only by putting on a break on the actual will you see the reason why it is running red:



In any event, here is the test explorer with at least one unit test for each of the functions that were just added.

[TestMethod]

public void PowerUsingValidData\_ReturnsExpected()

{

var calculations = new Calculations();

Double expected = 8;

Double actual = Math.Round(calculations.Power(2.0, 3.0), 0);

Assert.AreEqual(expected, actual);

}

[TestMethod]

public void CumulativeDistributionUsingValidData\_ReturnsExpected()

{

var calculations = new Calculations();

Double expected = .84134;

Double actual = Math.Round(calculations.CumulativeDistribution(1.0),5);

Assert.AreEqual(expected, actual);

}

[TestMethod]

public void BlackScholesCallUsingValidData\_ReturnsExpected()

{

var calculations = new Calculations();

Double expected = 4.4652;

var inputData = new BlackScholesInputData(58.6, 60.0, .5, .01, .3);

Double actual = Math.Round(calculations.BlackScholes(inputData,PutCallFlag.Call), 5);

Assert.AreEqual(expected, actual);

}

[TestMethod]

public void BlackScholesPutUsingValidData\_ReturnsExpected()

{

var calculations = new Calculations();

Double expected = 5.56595;

var inputData = new BlackScholesInputData(58.6, 60.0, .5, .01, .3);

Double actual = Math.Round(calculations.BlackScholes(inputData, PutCallFlag.Put), 5);

Assert.AreEqual(expected, actual);

}

[TestMethod]

public void DaysToYearsUsingValidData\_ReturnsExpected()

{

var calculations = new Calculations();

Double expected = .08214;

Double actual = Math.Round(calculations.DaysToYears(30), 5);

Assert.AreEqual(expected, actual);

}

[TestMethod]

public void BlackScholesDeltaCallUsingValidData\_ReturnsExpected()

{

var calculations = new Calculations();

Double expected = .50732;

var inputData = new BlackScholesInputData(58.6, 60.0, .5, .01, .3);

Double actual = Math.Round(calculations.BlackScholesDelta(inputData, PutCallFlag.Call), 5);

Assert.AreEqual(expected, actual);

}

[TestMethod]

public void BlackScholesDeltaPutUsingValidData\_ReturnsExpected()

{

var calculations = new Calculations();

Double expected = -.49268;

var inputData = new BlackScholesInputData(58.6, 60.0, .5, .01, .3);

Double actual = Math.Round(calculations.BlackScholesDelta(inputData, PutCallFlag.Put), 5);

Assert.AreEqual(expected, actual);

}

[TestMethod]

public void BlackScholesGammaUsingValidData\_ReturnsExpected()

{

var calculations = new Calculations();

Double expected = .39888;

var inputData = new BlackScholesInputData(58.6, 60.0, .5, .01, .3);

Double actual = Math.Round(calculations.BlackScholesGamma(inputData), 5);

Assert.AreEqual(expected, actual);

}

[TestMethod]

public void BlackScholesVegaUsingValidData\_ReturnsExpected()

{

var calculations = new Calculations();

Double expected = 16.52798;

var inputData = new BlackScholesInputData(58.6, 60.0, .5, .01, .3);

Double actual = Math.Round(calculations.BlackScholesVega(inputData), 5);

Assert.AreEqual(expected, actual);

}

[TestMethod]

public void BlackScholesThetaCallUsingValidData\_ReturnsExpected()

{

var calculations = new Calculations();

Double expected = -5.21103;

var inputData = new BlackScholesInputData(58.6, 60.0, .5, .01, .3);

Double actual = Math.Round(calculations.BlackScholesTheta(inputData, PutCallFlag.Call), 5);

Assert.AreEqual(expected, actual);

}

[TestMethod]

public void BlackScholesThetaPutUsingValidData\_ReturnsExpected()

{

var calculations = new Calculations();

Double expected = -4.61402;

var inputData = new BlackScholesInputData(58.6, 60.0, .5, .01, .3);

Double actual = Math.Round(calculations.BlackScholesTheta(inputData, PutCallFlag.Put), 5);

Assert.AreEqual(expected, actual);

}

[TestMethod]

public void BlackScholesRhoCallUsingValidData\_ReturnsExpected()

{

var calculations = new Calculations();

Double expected = 12.63174;

var inputData = new BlackScholesInputData(58.6, 60.0, .5, .01, .3);

Double actual = Math.Round(calculations.BlackScholesRho(inputData, PutCallFlag.Call), 5);

Assert.AreEqual(expected, actual);

}

[TestMethod]

public void BlackScholesRhoPutUsingValidData\_ReturnsExpected()

{

var calculations = new Calculations();

Double expected = -17.21863;

var inputData = new BlackScholesInputData(58.6, 60.0, .5, .01, .3);

Double actual = Math.Round(calculations.BlackScholesRho(inputData, PutCallFlag.Put), 5);

Assert.AreEqual(expected, actual);

}

[TestMethod]

public void PriceAtMaturityUsingValidData\_ReturnsExpected()

{

var calculations = new Calculations();

Double expected = 480.36923;

var inputData = new MonteCarloInputData(58.6, 60.0, .5, .01, .3);

Double actual = Math.Round(calculations.PriceAtMaturity(inputData, 10.0), 5);

Assert.AreEqual(expected, actual);

}

[TestMethod]

public void MonteCarloUsingValidData\_ReturnsExpected()

{

var calculations = new Calculations();

var inputData = new MonteCarloInputData(58.6, 60.0, .5, .01, .3);

var random = new System.Random();

List<Double> randomData = new List<double>();

for (int i = 0; i < 1000; i++)

{

randomData.Add(random.NextDouble());

}

Double actual = Math.Round(calculations.MonteCarlo(inputData, randomData), 5);

var greaterThanFour = actual > 4.0;

var lessThanFive = actual < 5.0;

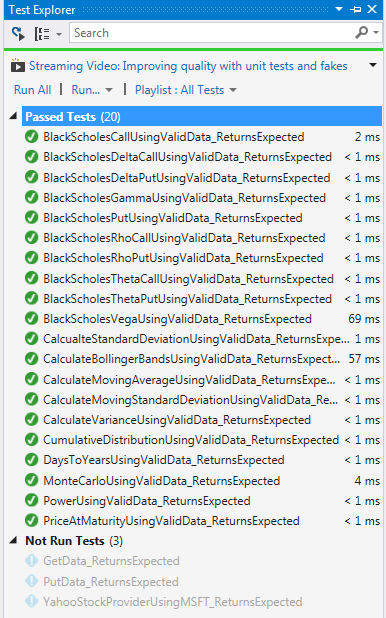
Assert.AreEqual(true, greaterThanFour);

Assert.AreEqual(true, lessThanFive);

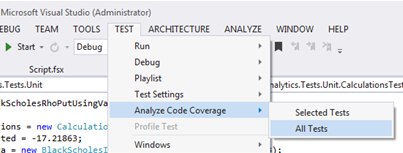
}

Note that the Monte Carlo function takes in an array of random values so the Assert cannot be AreEqual to an exact value. To get around this, there are two asserts for the range of values. Also note that the Monte Carlo method injects in the random array (versus hard-coding it into the function). This is a great design decision because if you enter in the same array over and over, the function will always return the same result. Hard-coded System.Random in functions make that function untestable.

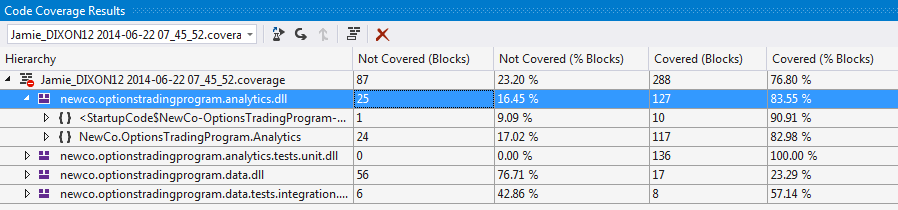
Once you run the tests, you should get this:



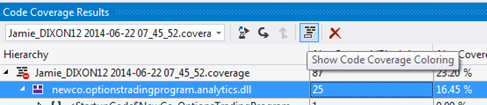
Once your tests run green, select the TEST -> Analyze Code Coverage -> All Tests menu item (Visual Studio Premium or Ultimate)



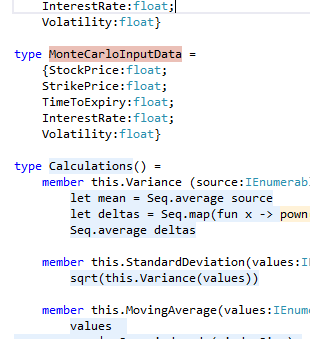
And you can see there is pretty good code coverage



If you select the “Show Code Coverage Coloring” button on the Code Coverage Results window



You can see that the F# code gets this feature in Visual Studio



Which is great.

**Implement Non-Basic Statistics On The UI**

As or last step, we are going to put these statistics on our UI. Open the NewCo.OptionsTradingProgram.UI.Desktop project and navigate to the MainWindow.xaml file. Add in two new grids below the Bollinger Bands chart:

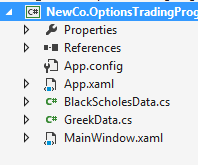
<DataGrid x:Name="TheGreeksDataGrid" HorizontalAlignment="Left" Height="206" Margin="10,10,0,0"

VerticalAlignment="Top" Width="483" Grid.Row="1" Grid.RowSpan="2"/>

<DataGrid x:Name="BlackScholesDataGrid" HorizontalAlignment="Left" Height="206" Margin="10,10,0,0"

VerticalAlignment="Top" Width="483" Grid.Row="3"/>

Add two new classes to the UI project (this would be the ‘M’ in your MVVM) called BlackScholesData and GreekData



And their implementation:

public class BlackScholesData

{

public Double StrikePrice { get; set; }

public Double Call { get; set; }

public Double Put { get; set; }

public Double MonteCarlo { get; set; }

}

public class GreekData

{

public Double StrikePrice { get; set; }

public Double DeltaCall { get; set; }

public Double DeltaPut { get; set; }

public Double Gamma { get; set; }

public Double Vega { get; set; }

public Double ThetaCall { get; set; }

public Double ThetaPut { get; set; }

public Double RhoCall { get; set; }

public Double RhoPut { get; set; }

}

Next, go to the MainWindow.caml.cs code behind and go into the GoButton\_Click envent handler and add the following code at the end of the function (below the graph code):

var theGreeks = new List<GreekData>();

for (int i = 0; i < 5; i++)

{

var greekData = new GreekData();

greekData.StrikePrice = closestDollar - i;

theGreeks.Add(greekData);

greekData = new GreekData();

greekData.StrikePrice = closestDollar + i;

theGreeks.Add(greekData);

}

theGreeks.Sort((greek1,greek2)=>greek1.StrikePrice.CompareTo(greek2.StrikePrice));

foreach (var greekData in theGreeks)

{

var inputData =

new BlackScholesInputData(adjustedClose, greekData.StrikePrice, .5, .01, .3);

greekData.DeltaCall = calculations.BlackScholesDelta(inputData, PutCallFlag.Call);

greekData.DeltaPut = calculations.BlackScholesDelta(inputData, PutCallFlag.Put);

greekData.Gamma = calculations.BlackScholesGamma(inputData);

greekData.RhoCall = calculations.BlackScholesRho(inputData, PutCallFlag.Call);

greekData.RhoPut = calculations.BlackScholesRho(inputData, PutCallFlag.Put);

greekData.ThetaCall = calculations.BlackScholesTheta(inputData, PutCallFlag.Call);

greekData.ThetaPut = calculations.BlackScholesTheta(inputData, PutCallFlag.Put);

greekData.Vega = calculations.BlackScholesVega(inputData);

}

this.TheGreeksDataGrid.ItemsSource = theGreeks;

var blackScholes = new List<BlackScholesData>();

for (int i = 0; i < 5; i++)

{

var blackScholesData = new BlackScholesData();

blackScholesData.StrikePrice = closestDollar - i;

blackScholes.Add(blackScholesData);

blackScholesData = new BlackScholesData();

blackScholesData.StrikePrice = closestDollar + i;

blackScholes.Add(blackScholesData);

}

blackScholes.Sort((bsmc1, bsmc2) => bsmc1.StrikePrice.CompareTo(bsmc2.StrikePrice));

var random = new System.Random();

List<Double> randomData = new List<double>();

for (int i = 0; i < 1000; i++)

{

randomData.Add(random.NextDouble());

}

foreach (var blackScholesMonteCarlo in blackScholes)

{

var blackScholesInputData =

new BlackScholesInputData(adjustedClose, blackScholesMonteCarlo.StrikePrice, .5, .01, .3);

var monteCarloInputData =

new MonteCarloInputData(adjustedClose, blackScholesMonteCarlo.StrikePrice, .5, .01, .3);

blackScholesMonteCarlo.Call = calculations.BlackScholes(blackScholesInputData, PutCallFlag.Call);

blackScholesMonteCarlo.Put = calculations.BlackScholes(blackScholesInputData, PutCallFlag.Put);

blackScholesMonteCarlo.MonteCarlo = calculations.MonteCarlo(monteCarloInputData, randomData);

}

this.BlackScholesDataGrid.ItemsSource = blackScholes;

And now Run your UI

