GLOBAL eSOLUTIONS

AUTON Algoscript

Version 1.0
Developer's Guide

February 10, 2017
Version 1.00

Disclaimer: This work is the property Global eSolutions (HK) Limited (the "Company" or "GES"). Permission is granted for this material to be shared for non-commercial, educational purpose, provided that this copyright statement appears on the reproduced materials and notice is given that the copying is by permission of the Company. To disseminate otherwise or to republish requires written permission of the Company.

"AUTON" and "AUTON Algoscript" are trademarks of the Company or its affiliates.

All other trademarks and registered trademarks are the property of their respective owners.

Global eSolutions (HK) Limited
Room 3004-3008, 30/F, Shui On Centre, 6-8 Harbour Road, Wan Chai, Hong Kong
香港灣仔港灣道六至八號瑞安中心三十樓三零零四至三零零八室

T: +852 3412 3600
F: +852 2851 0017
W: www.ges.com.hk

1 Introduction to AUTON Algoscript

1.1 Overview of AUTON Algoscript

AUTON Algoscript is a proprietary C#-based script language used for directing GES's AUTON trading platform into making trade execution based on various user-defined conditions. AUTON Algoscript lets you integrate streaming real-time and delayed data, historical data, technical indicators, and even custom data into your own trading applications. It uses an event-driven model, which is triggered on the event on new price data. The user may program his/her own time-based event triggers should he/she wishes to do so.

The script supports a host of Windows .NET libraries for maximum flexibility for the user. It can be executed in live mode for real trade execution, or in back-test mode based on historical price data provided by AUTON or the users themselves. AUTON's reporting interface will generate a host of relevant statistical data regarding the performance of the user's trading strategy.

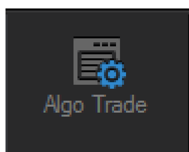
In short, AUTON Algoscript is a flexible, state-of-the-art programming platform for developing automated trading system, included as part of every copy of AUTON client.

2 Algoscript Programming

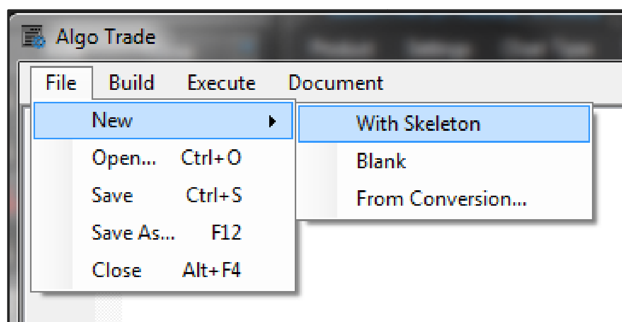
Algoscript programming is the core feature of the AUTON system. The flexibility of a C#-based architecture enables Algoscript to not only support execution of trades, but also user-built risk management platform, network infrastructure, research suites, and portfolio optimization tools. The possible implementations of tools are endless and limited only to the user's level of imagination and skills. The following tutorial will allow a new user to get familiar with the Algoscript programming language with an in-built example.

2.1 Creating your first Algoscript file

All Algoscript file has extension .alg, which upon compilation by AUTON will create an Algoscript executable file with extension .ale. To create a new Algoscript file, click on the Algo Trade button in the AUTON main menu:



The Algo Trade window will now pop up. Click File -> New -> With Skeleton:



You have created your first Algoscript file, we will now examine below the structure of a basic Algoscript file and the purpose of each of its functions.

2.2 Structure of an Algoscript File

The Algoscript file is essentially a C# class comprised of several major components:

```
1   using System;
2   using AlgoTrade.Interface;
3
4   namespace AlgoTrade
5   {
6       public class RUN_ALGO_NAME : Program
7       {
8
9           public RUN_ALGO_NAME (Interface.IAlgoTradeFunction func) : base(func)
10          {
11
12          }
13
14          /**
15              Start() will be called when using script mode
16          **/
17          public override int Start ()
18          {
19              // Start your implementation here
20              return 0 ;
21          }
22
23          /**
24              OnTick() will be called when using Run With ... mode
25          **/
26          public override int OnTick()
27          {
28              return 0;
29          }
30      }
31  }
32
```

Declaration

Constructor

Start() (Used only in script mode)

OnTick() (Runs when a price tick comes)

The default class name "RUN_ALGO_NAME" should be replaced by the name of your algorithm.

The space labeled as ***Declaration*** is used to define any Input Parameters and/or public/private variables you may use within the script.

The ***Constructor*** is used to assign values to any variables that are declared in the Declaration section prior to execution. Again, the default constructor name "RUN_ALGO_NAME" should be replaced by the name of your algorithm.

Start should only be used in script mode, the function is called only once at the start of the script execution. In other mode it is essentially the same as ***OnTick*** and thus can be ignored.

OnTick executes when a new price tick has arrived into the system, and is the key function developers will spend the most time on.

2.3 A Simple Long-Only Momentum Algorithm

Let us begin by developing a simple momentum algorithm: If the previous bar's CLOSE is less than the current bar's CLOSE, open a pre-defined number of lots BUY position in the Selected Product at Market Price. Vice versa, if the previous bar's CLOSE is higher than the current bar's CLOSE, settle any outstanding BUY position we have at Market Price. Also, let's rename our algorithm to "LongOnlyMomentumAlgo".

Steps to program:

1. Replace RUN_ALGO_NAME with LongOnlyMomentumAlgo
2. Add an input parameter, "Lots" which defines the number of lots we like our algorithm to open in the condition of opening a buy position
3. On every tick, check if the current bar has changed. If so, check if the new bar's CLOSE, compared to the previous bar's CLOSE, requires us to enter or exit a position.
4. Handle the relevant trade actions as required

Completing steps 1 and 2, our Declaration and Constructor section of our algorithm become this:

```
public class LongOnlyMomentumAlgo : Program
{
    [InputParameter] public MDouble Lots = 1.0;

    public LongOnlyMomentumAlgo (Interface.IAlgoTradeFunction func) : base(func)
    {

    }
…
```

The ***[InputParameter]*** tag is unique to Algoscript and is used to identify the variables that take the user's input. The input variables must be public and their type must starts with a prefix "M", the variable types that are supported include:

- **MDouble**, the Algoscript implementation of the double C# element
- **MInt**, the Algoscript implementation of the int C# element
- **MString**, the Algoscript implementation of the string C# element
- **MDateTime**, the Algoscript implementation of the DateTime C# object

GLOBAL eSOLUTIONS

In this case an input *double* by the name "Lots" was declared with a default value of 1.0.

We are now ready to engage steps 3 and 4. Our trading logic dictates that the current bar's CLOSE should be compared with the previous bar's CLOSE to determine whether we should make a trade. However, because the OnTick function will be triggered on every tick price change instead of every bar change, we would need a method to determine whether a bar has changed when the OnTick function was triggered.

To do this, we will make use of an AUTON inbuilt function GetChartDataTime(), the definition of which can be accessed from AUTON Algoscript panel by clicking on the Document->API Document menu option:

Program.MDateTime AlgoTrade.Interface.IAlgoTradeFunction.GetChartDataTime (Program.MString product, Program.MInt period, Program.MInt shift)

Returns Time value for the bar of indicated product with period and shift. If local history is empty (not loaded), function returns 0.

Parameters

product	GetProductDesc the data of which should be used to calculate study. NULL means the current product.
period	period. It can be any of period enumeration values. 0 means the current chart period.
shift	Index of the value taken from the study buffer (shift relative to the current bar the given amount of periods ago).

Returns

Time

The MDateTime of the latest bar can be accessed by:

```
MDateTime dtCurrentTime = GetChartDataTime(null, 0, 0);
```

If the MDateTime of the current bar has changed from our last check, then there is a new bar of price data (which is refreshed every period). Knowing this, we set a private DateTime? variable in our Declaration section, dtLastBarTime, to keep track of our latest bar time:

```
private DateTime? dtLastbarTime = null;
```

Within the OnTick function, add an if statement that determines whether if the bar has changed:

```
MDateTime dtCurrentTime = GetChartDataTime(null, 0, 0);
If (dtLastBarTime == null || dtLastBarTime.Value != dtCurrentTime.csDatetime)
{
    dtLastBarTime = dtCurrentTime.csDatetime;
    // Rest of the trade logic goes here
}
else
```

```
    return 0;
```

The MDateTime.csDatetime parameter is used to return a C# DateTime object given an MDateTime.

Now on to the meat of the algorithm; to compare the CLOSE of the previous two bars, use the functions:

```
MDouble numClose = GetChartDataClose(null, 0, 1);
MDouble numPrevClose = GetChartDataClose(null, 0, 2);
```

Furthermore, the function to open and close an order:

```
MInt nTicket = SubmitOpenOrder(GetProductDesc(), Const.OP_BUY, Lots, Ask, 0, 0, 0, "Long Momentum", 0, null, null);
MBool bSuccess = SubmitCloseOrder(nTicket, Lots, Bid, 0, null);
```

This requires adding two extra variables in the Declaration section to keep track of the outstanding order:

```
private int nTicket = 0;
private bool bOutstanding = false;
```

We are now ready to code the OnTick function, as shown below:

```
public void OnTick()
{
    double numBid = GetCurrentQuote(GetProductDesc(), Const.MODE_BID);
    double numAsk = GetCurrentQuote(GetProductDesc(), Const.MODE_ASK);

    MDateTime dtCurrentTime = GetChartDataTime(null, 0, 0);
    If (dtLastBarTime == null | | dtLastBarTime.Value != dtCurrentTime.csDateTime)
    {
        dtLastBarTime = dtCurrentTime.csDateTime;
        MDouble numClose = GetChartDataClose(null, 0, 1);
        MDouble numPrevClose = GetChartDataClose(null, 0, 2);

        // Buy
        if (numClose > numPrevClose)
        {
            if (!bOutstanding)
            {
                nTicket = SubmitOpenOrder(GetProductDesc(), Const.OP_BUY, Lots,
            numAsk, 0, 0,
                    0, "Long Momentum", 0, null, null);
                bOutstanding = true;
            }
        }
        // Settle Buy, if there is an order outstanding
        else
        {
            if (bOutstanding)
            {
                bool bSuccess = SubmitCloseOrder(nTicket,  Lots, numBid, 0, null);
```

```
                if (bSuccess)
                    bOutstanding = false;
                else
                    Print("Failed to SETTLE BUY: " + GetLastExecuteError());
            }
        }
    }
    else
        return 0;
    return 0;
}
```

The SubmitOpenOrder function returns a ticket number by which characteristic your algorithm can use determine the status of the order:

- If nTicket < 0: The order is pending for dealer acceptance
- If nTicket = 0: The order is rejected by the dealer
- If nTicket > 0: The order is accepted by the dealer

The ideal case is of course the order being immediately accepted by the dealer; in such case we will set the status of the "bOutstanding" bool value to true, to indicate that an outstanding position is currently being held by the system. In the case of an order rejection, the trade is simply skipped without setting "bOutstanding" to true.

The more difficult case is when the order is pending for the dealer's acceptance. Here the system will return with a temporary negative ticket number, the order either upon acceptance by the dealer will turn into an order with a positive ticket number or upon rejection will simply disappear from the system. However, retrieving the detail of any order requires first selecting it with the ticket number; with the ticket number either changing or disappearing, the order can no longer be referenced and we wouldn't know whether it was accepted or rejected.

The solution to this problem requires the creation of a function to check for the status of the ticket given a unique identifier number (magic number) was set on the original order when sent. Therefore we add an additional input field to our algorithm in the Declaration section:

```
[InputParameter]  public MInt MagicNumber = 10001;
private bool bPending = false;
```

We make a modification to our OnTick function to check if there is any pending order and to add in the input of the MagicNumber to our SubmitOpenOrder call:

```
public void OnTick()
{
    CheckPendingOrderFill();

    double numBid = GetCurrentQuote(GetProductDesc(), Const.MODE_BID);
    double numAsk = GetCurrentQuote(GetProductDesc(), Const.MODE_ASK);

    MDateTime dtCurrentTime = GetChartDataTime(null, 0, 0);
    If (dtLastBarTime == null || dtLastBarTime.Value != dtCurrentTime.csDateTime)
    {
```

```
            dtLastBarTime = dtCurrentTime.csDateTime;
            MDouble numClose = GetChartDataClose(null, 0, 1);
            MDouble numPrevClose = GetChartDataClose(null, 0, 2);

            // Buy
            if (numClose > numPrevClose)
            {
                if (!bOutstanding && !bPending)
                {
                    nTicket = SubmitOpenOrder(GetProductDesc(), Const.OP_BUY, Lots,
                numAsk, 0, 0, 0, "Long Momentum", MagicNumber, null, null);
                    if (nTicket > 0)
                    {
                        Print("OPEN BUY Order: Accepted");
                        bOutstanding = true;
                    }
                    else if (nTicket < 0)
                    {
                        Print("OPEN BUY Order: Dealer Pending");
                        bPending = true;
                    }
                    else if (nTicket == 0)
                    {
                        Print("OPEN BUY Order: Rejected");
                    }
                }
            }
            // Settle Buy, if there is an order outstanding
            else
            {
                if (bOutstanding)
                {
                    bool bSuccess = SubmitCloseOrder(nTicket,  Lots, numBid, 0, null);
                    if (bSuccess)
                        bOutstanding = false;
                    else
                        Print("Failed to SETTLE BUY: " + GetLastExecuteError());
                }
            }
        }
        else
            return 0;
}
```

Finally, we implement the CheckPendingOrderFill function:

```
private void CheckPendingOrderFill()
{
    if (!bPending)
        return;

    // Pending outstanding OPEN BUY
    for (int i = 0; i < GetOutstandingOrderCount(); i++)
```

```
    {
        if (!SelectOrder(i, Const.SELECT_BY_POS, Const.MODE_TRADES))
            break;
        if (GetOrderID() == null | | GetOrderID() != MagicNumber)
            continue;
        if (GetOrderTicket() > 0)
        {
            nTicket = GetOrderTicket();
            Print("Pending OPEN BUY Order: Accepted");
            bOutstanding = true;
            bPending = false;
            return;
        }
        else if (GetOrderTicket() == 0)
        {
            nTicket = 0;
            Print("Pending OPEN BUY Order: Rejected");
            bPending = false;
            return;
        }
    }

    // Order disappeared, treat as rejected
    nTicket = 0;
    Print("Pending OPEN BUY Order: Rejected");
    bPending = false;
    return;
}
```

The algorithm is now completed.

```
using System;

using AlgoTrade.Interface;



namespace AlgoTrade

{

    public class LongOnlyMomentumAlgo : Program

    {

        [InputParameter] public MDouble Lots = 1.0;

        [InputParameter] public MInt MagicNumber = 10001;

        private DateTime? dtLastBarTime = null;

        private int nTicket = 0;

        private bool bOutstanding = false;
```

GLOBAL eSOLUTIONS

```
private bool bPending = false;

public LongOnlyMomentumAlgo (Interface.IAlgoTradeFunction func) : base(func)
{

}

/**
    Start() will be called when using script mode
**/
public override int Start ()
{
    // Start your implementation here
    return 0 ;
}

/**
    OnTick() will be called when using Run With ... mode
**/
public override int OnTick()
{
    CheckPendingOrderFill();

    double numBid = GetCurrentQuote(GetProductDesc(), Const.MODE_BID);
    double numAsk = GetCurrentQuote(GetProductDesc(), Const.MODE_ASK);

    MDateTime dtCurrentTime = GetChartDataTime(null, 0, 0);
```

Global eSolutions (HK) Limited
Room 3004-3008, 30/F, Shui On Centre, 6-8 Harbour Road, Wan Chai, Hong Kong
香港灣仔港灣道六至八號瑞安中心三十樓三零零四至三零零八室
T: +852 3412 3600
F: +852 2851 0017
W: www.ges.com.hk

```
        if (dtLastBarTime == null || dtLastBarTime.Value != dtCurrentTime.csDatetime)
        {
            dtLastBarTime = dtCurrentTime.csDatetime;

            // Rest of the trade logic goes here
            MDouble numClose = GetChartDataClose(null, 0, 1);
            MDouble numPrevClose = GetChartDataClose(null, 0, 2);

            // Buy
            if (numClose > numPrevClose)
            {
                if (!bOutstanding && !bPending)
                {
                    nTicket = SubmitOpenOrder(GetProductDesc(), Const.OP_BUY, Lots,
numAsk,
                        0, 0, 0, "Long Momentum", MagicNumber, null, null);
                    if (nTicket > 0)
                    {
                        Print("OPEN BUY Order: Accepted");
                        bOutstanding = true;
                    }
                    else if (nTicket < 0)
                    {
                        Print("OPEN BUY Order: Dealer Pending");
                        bPending = true;
                    }
                    else if (nTicket  == 0)
                    {
```

```
                        Print("OPEN BUY Order: Rejected");
                    }
                }
            }
            // Settle Buy, if there is an order outstanding
            else
            {
                if (bOutstanding)
                {
                    bool bSuccess = SubmitCloseOrder(nTicket, Lots, numBid, 0, null);
                    if (bSuccess)
                        bOutstanding = false;
                    else
                        Print("Failed to SETTLE BUY (" + nTicket + "): " +
GetLastExecuteError());
                }
            }
        }
        else
            return 0;

        return 0;
    }

    private void CheckPendingOrderFill()
    {
        if (!bPending)
            return;
```

```
// Pending outstanding OPEN BUY
for (int i = 0; i < GetOutstandingOrderCount(); i++)
{
    if (!SelectOrder(i, Const.SELECT_BY_POS, Const.MODE_TRADES))
        break;

    Print("Order " + i + ": " + GetOrderID());

    if (GetOrderID() == null || GetOrderID() != MagicNumber)
        continue;
    if (GetOrderTicket() > 0)
    {
        nTicket = GetOrderTicket();
        Print("Pending OPEN BUY Order: Accepted");
        bOutstanding = true;
        bPending = false;
        return;
    }
    else if (GetOrderTicket() == 0)
    {
        nTicket = 0;
        Print("Pending OPEN BUY Order: Rejected");
        bPending = false;
        return;
    }
}
```

```
                // Order disappeared, treat as rejected
                nTicket = 0;
                Print("Pending OPEN BUY Order: Rejected");
                bPending = false;
                return;
            }
        }
}
```

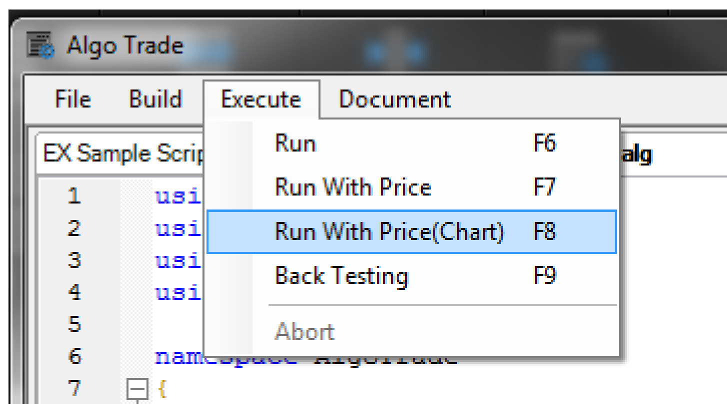
3 Executing the Algoscript

AUTON supports four ways in which an Algoscript file can be run:

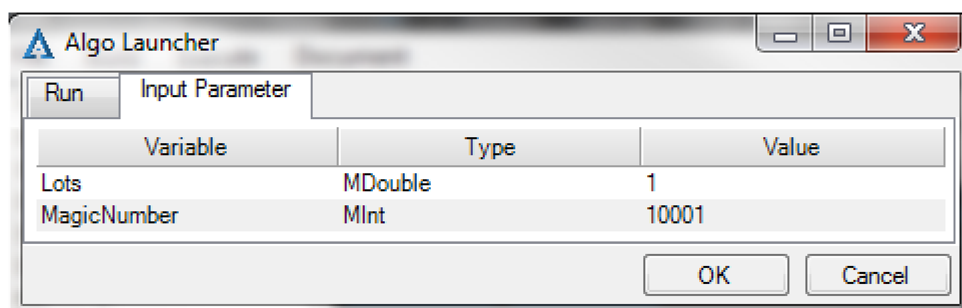
1. Run – Executing the algorithm as a script
2. Run With Price – Attaching the algorithm to a security and executes it using live price
3. Run With Price (Chart) – Invoke a chart window and executes the algorithm using live price
4. Back Testing – Invoke the back-tester which executes the algorithm using historical price

3.1 Running the Algoscript in Live Mode

Let's attach the script we created in the previous section to a chart for execution. Click on the menu option Run -> Run With Price (Chart).

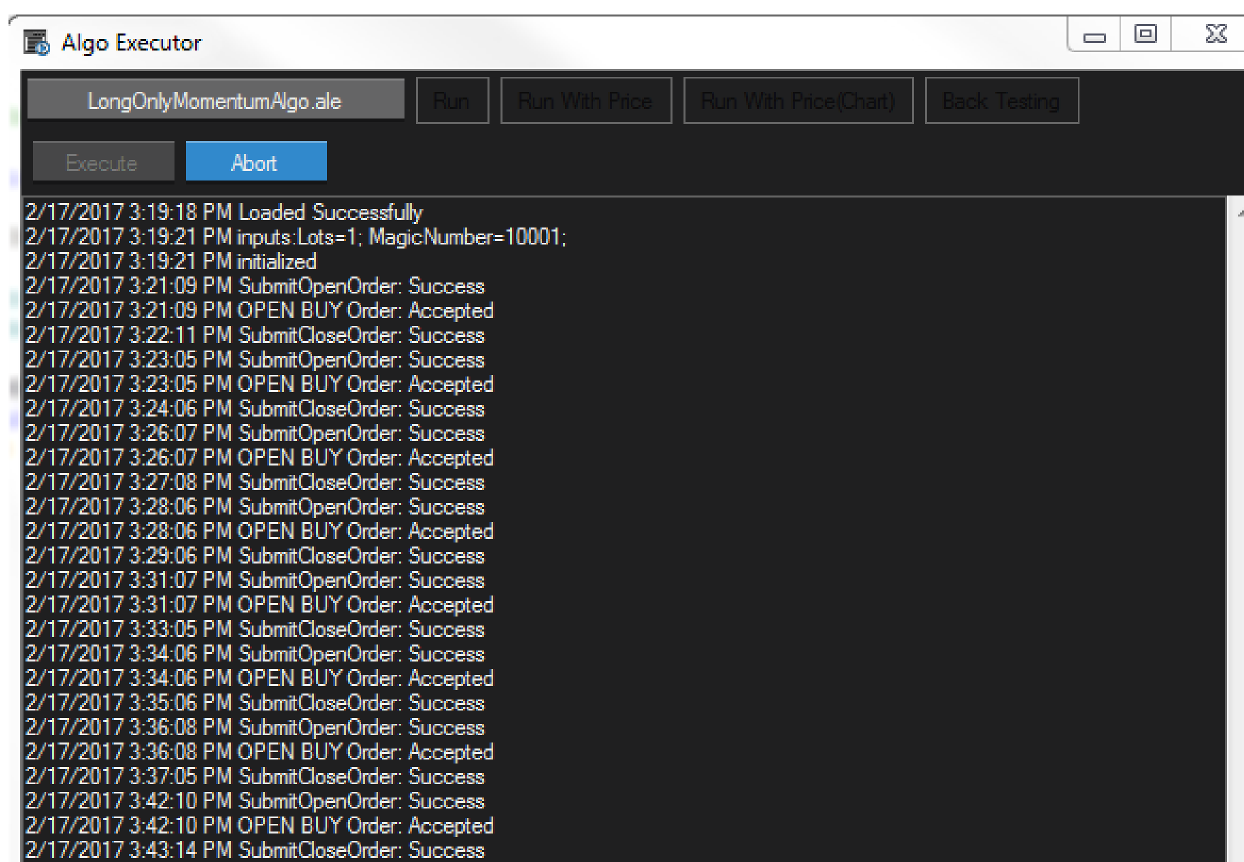


A menu pops up to enable the user to input the trade product and the chart time interval. For the time being, select the product 00005.HK (HSBC Holdings) and a chart interval of 1 minute with 2000 points of data. Click on the Input Parameter Tab.



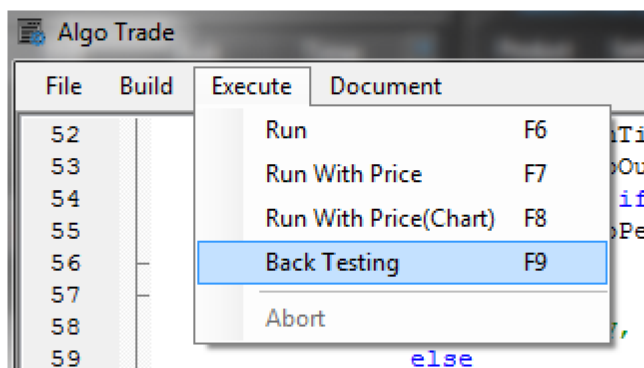
Noticed the variables we designated as [InputParameter] are shown here, we can leave the traded lots to 1 and Magic Number to 10001. Press OK to start the algorithm.

The Algo Executor will keep track of any trades the algorithm made based on the embedded Print statements. As expected, the algorithm open trades whenever there is an uptick in price.



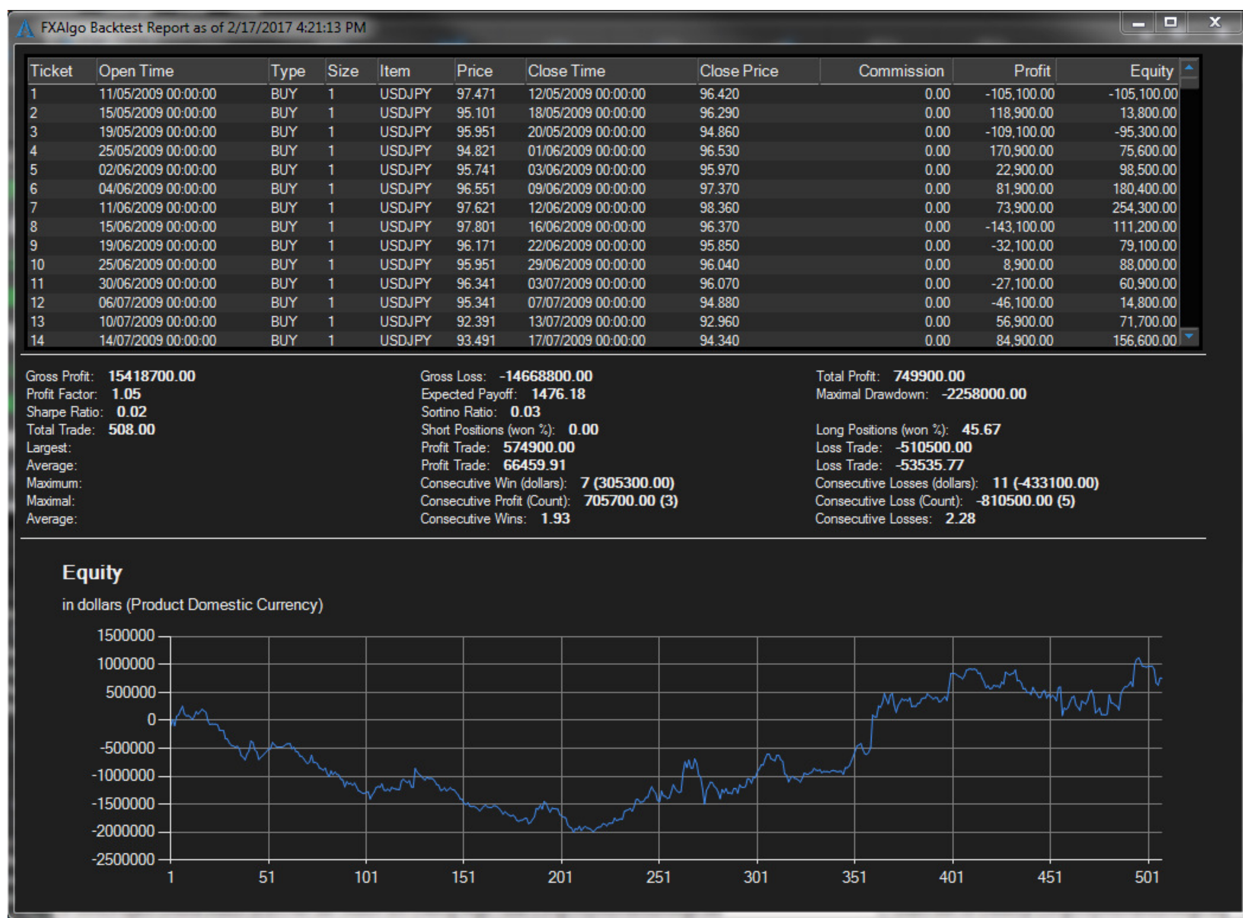
3.2 Running the Algoscript in Back Testing Mode

We can conduct a full back-test on the script using the AUTON in-built back-testing suite. Click on menu option Run -> Back Testing.



The Back Testing menu is similar to that of Run With Price; we will test this strategy against the USDJPY currency pair using 2000 points of daily price data, with commission per lot and spread set to zero for demonstration purpose.

Click OK to start the back test.



Ticket	Open Time	Type	Size	Item	Price	Close Time	Close Price	Commission	Profit	Equity
1	11/05/2009 00:00:00	BUY	1	USDJPY	97.471	12/05/2009 00:00:00	96.420	0.00	-105,100.00	-105,100.00
2	15/05/2009 00:00:00	BUY	1	USDJPY	95.101	18/05/2009 00:00:00	96.290	0.00	118,900.00	13,800.00
3	19/05/2009 00:00:00	BUY	1	USDJPY	95.951	20/05/2009 00:00:00	94.860	0.00	-109,100.00	-95,300.00
4	25/05/2009 00:00:00	BUY	1	USDJPY	94.821	01/06/2009 00:00:00	96.530	0.00	170,900.00	75,600.00
5	02/06/2009 00:00:00	BUY	1	USDJPY	95.741	03/06/2009 00:00:00	95.970	0.00	22,900.00	98,500.00
6	04/06/2009 00:00:00	BUY	1	USDJPY	96.551	09/06/2009 00:00:00	97.370	0.00	81,900.00	180,400.00
7	11/06/2009 00:00:00	BUY	1	USDJPY	97.621	12/06/2009 00:00:00	98.360	0.00	73,900.00	254,300.00
8	15/06/2009 00:00:00	BUY	1	USDJPY	97.801	16/06/2009 00:00:00	96.370	0.00	-143,100.00	111,200.00
9	19/06/2009 00:00:00	BUY	1	USDJPY	96.171	22/06/2009 00:00:00	95.850	0.00	-32,100.00	79,100.00
10	25/06/2009 00:00:00	BUY	1	USDJPY	95.951	29/06/2009 00:00:00	96.040	0.00	8,900.00	88,000.00
11	30/06/2009 00:00:00	BUY	1	USDJPY	96.341	03/07/2009 00:00:00	96.070	0.00	-27,100.00	60,900.00
12	06/07/2009 00:00:00	BUY	1	USDJPY	95.341	07/07/2009 00:00:00	94.880	0.00	-46,100.00	14,800.00
13	10/07/2009 00:00:00	BUY	1	USDJPY	92.391	13/07/2009 00:00:00	92.960	0.00	56,900.00	71,700.00
14	14/07/2009 00:00:00	BUY	1	USDJPY	93.491	17/07/2009 00:00:00	94.340	0.00	84,900.00	156,600.00

The back test report generated in the end of the execution gives the user an overview of all the trades that were made by the algorithm, as well as performance information including Gross Profit/Loss, Maximum Drawdown, Profit Factor, Sortino Ratio, Sharpe Ratio, number of trades, etc. The report also included an equity chart for visualization of your performance. This momentum strategy has proven to be profitable for this security (USDJPY) in the past when executed on daily bar price.

GLOBAL
eSOLUTIONS

4 Help and References

4.1 AlgoTrade API

Perhaps the best place to look for support on Algoscript programming is by the Document->API Document option menu within AlgoTrade window. The AlgoTrade API contains the definition of every functions supported by Algoscript and descriptions of their inputs/outputs.

4.2 GES Support Desk

For enquiries and support, please email us your questions to mkt@ges.com.hk and we will reply within 48 hours.