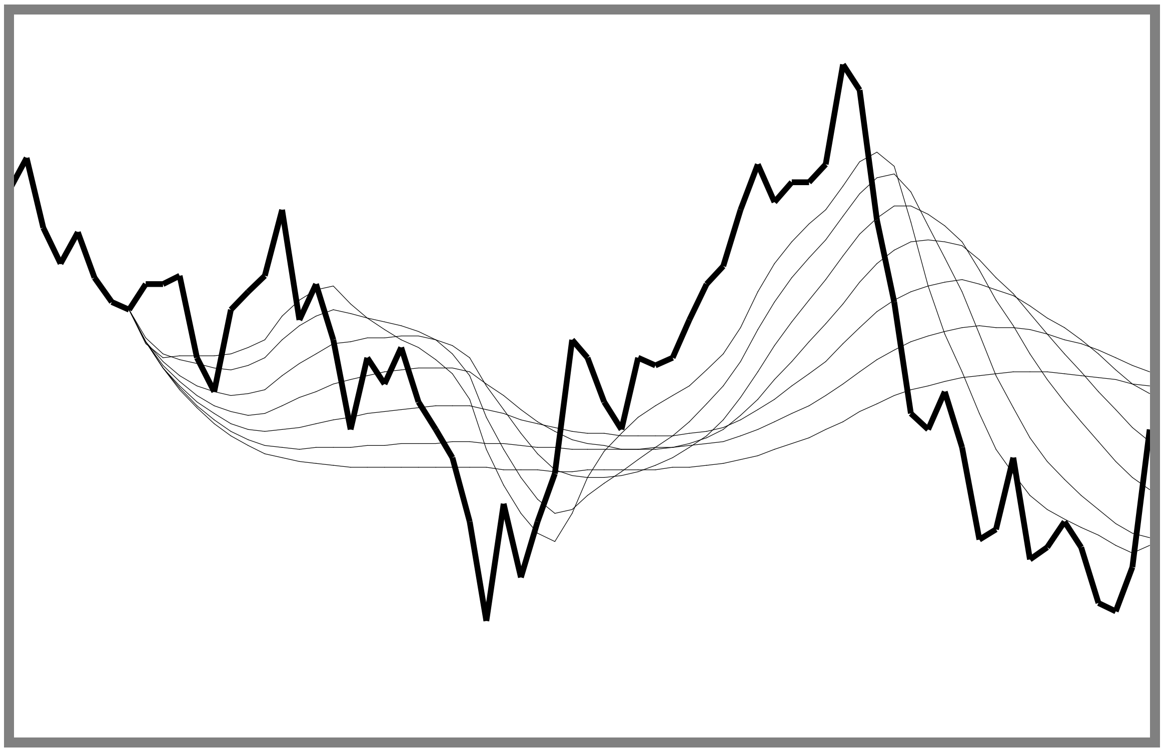
Jurik Research Limited Use Software License Agreement

JURIK RESEARCH

JMA

Jurik Moving Average DLL module
for Windows® Application Developers



USER'S GUIDE

Requirements

- Windows 98, 2000, NT4 or XP
- Application software that can access DLL functions

Installing the 32 bit DLL module

1. Execute the Installer, JRS_DLL.EXE. It will analyze your computer and give you a computer identification number. Write it down.
2. Get your access PASSWORD from Jurik Research Software. You can do so by calling 323-258-4860 (USA), faxing 323-258-0598 (USA), e-mailing support@nfsmith.net, or writing Jurik Research Software at 686 South Arroyo Parkway, Suite 237, Pasadena, California 91105. Be sure to give your full name, mailing address and computer identification number. You will then be given a password.
3. Rerun the installer JRS_DLL.EXE, this time entering the password when asked. Also enter **all the Jurik Research modules that you currently are licensed to run**. It will copy the latest version of these modules to any directory you specify.

You may now code your software to access the DLL as described on the following pages. First, read the important notices below.

!! IMPORTANT !!

ABOUT PASSWORDS

And what to do when they become invalid

If you upgrade to a new computer, or significantly upgrade your existing computer (such as flash a new BIOS), you should reinstall JMA and all other Jurik tools that are licensed for your computer. The installer will let you know if your current password is no longer valid. Also, if you want to run JMA on additional computers, you will need additional passwords. For new or replacement passwords, call 323-258-4860

ABOUT DATA VALIDITY

And what to do when JMA encounters an error

When JMA encounters a problem, (e.g. the password used during installation has become invalid), JMA will continue to run but the data produced will not be valid. To let you know this is the case, JMA will return an appropriate error code, bit it will NOT post any warning message on your monitor. Therefore …

Do not assume JMA will always smooth your data. You must validate JMA’s output by CHECKING THE RETURN ERROR CODE immediately after each call to JMA.

Why use **JMA ?**

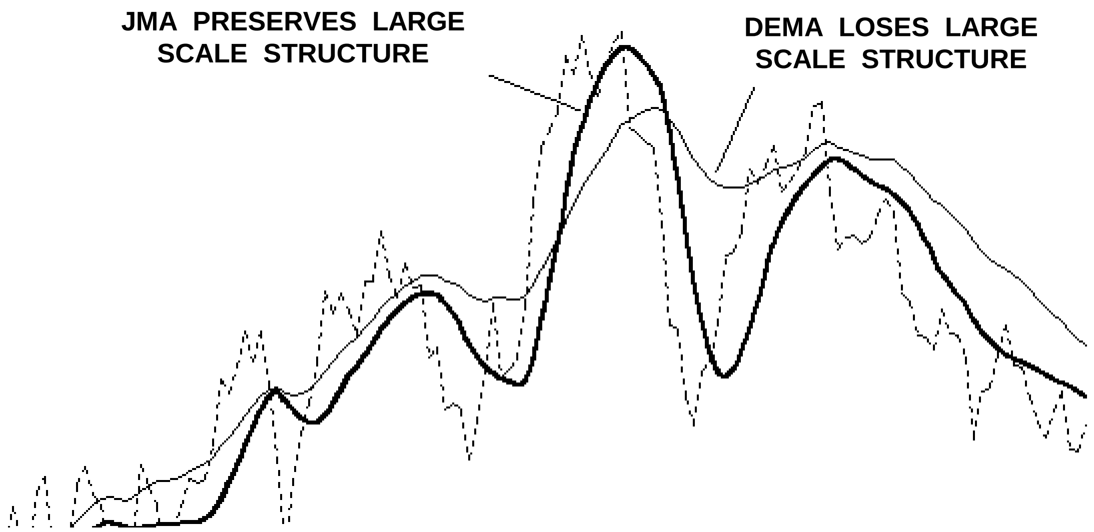
The ADAPTIVE MOVING AVERAGE by Jurik Research

Daily prices produce a time series with some amount of random fluctuations. To remove this noise, market technicians typically use moving average (MA) filters. Only Jurik's JMA excels in all four benchmarks of a truly great filter...

BENCHMARK #1: ACCURACY

Moving Average (MA) filters have an adjustable parameter that controls its speed. Speed governs two opposing properties of a filter: smoothness (lack of random zigzagging) and accuracy (closeness to the original data). That is, the smoother a filter becomes, the less it accurately resembles the original time series. This makes sense, since we do not want to accurately track zigzagging noise within our data.

Because the financial investor tries to apply just enough smoothness to filter out noise without removing important structure in price activity. For example, in the chart below, the popular Double Exponential Moving Average (DEMA) is just as smooth as JMA yet DEMA fails to track large scale structure (the big cycles). On the other hand, JMA follows the cyclic action very well.

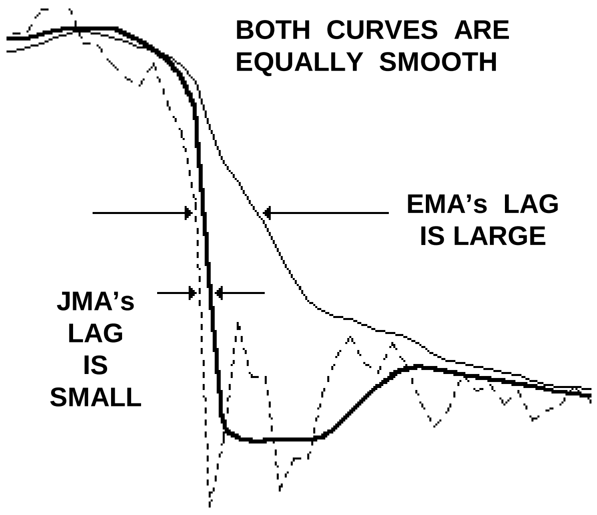


BENCHMARK #2: TIMELINESS

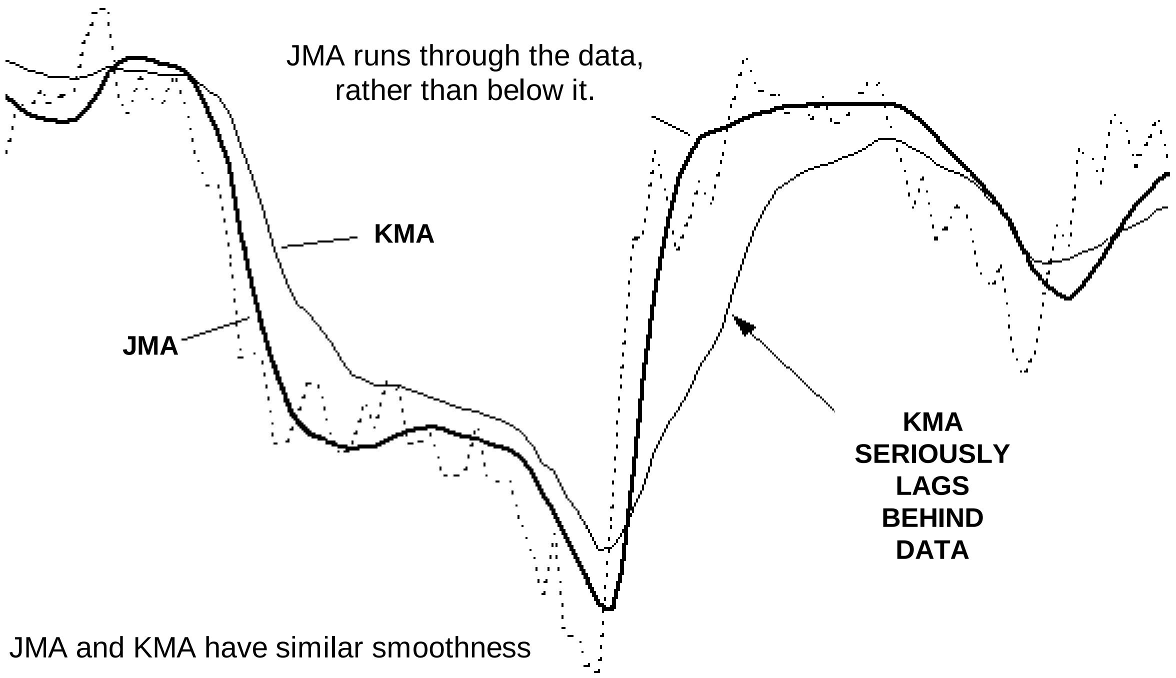
Most MA filters have another problem: they lag behind the original time series. This is a critical issue because excessive delay and late trades may reduce profits significantly.

Ideally, you would like a filtered signal to be both smooth and lag free. For many types of moving average filters, including the three classics (simple, weighted, and exponential), greater smoothness produces greater lag.

For example, in the chart to the right, price action is the dotted line. The exponential moving average, EMA, lags well behind JMA (thick solid line). As you can see, with EMA's excessive lag, you would have had to wait a long time before it returned to the price action. In contrast, JMA never left it!



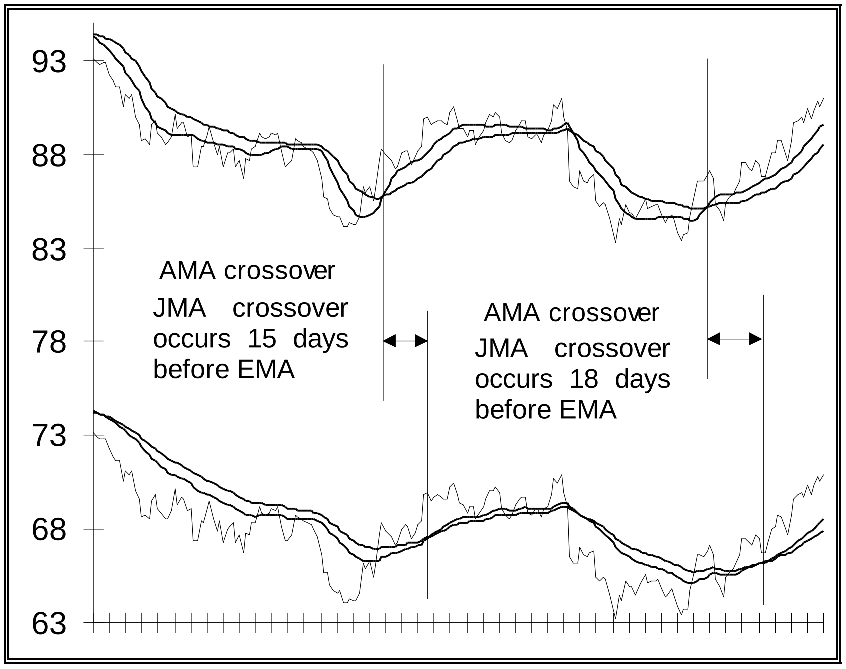
Adaptive filters developed by others, such as the Kaufman and Chande AMA, will also lag well behind your time series. Kaufman's Moving Average (KMA), is an exponential moving average whose speed is governed by the "efficiency" of price movement. For example, fast moving price with little retracement (a strong trend) is considered very efficient and the KMA will automatically speed up to prevent excessive lag. This interesting concept sometimes works well, sometimes not. For example, the chart below shows KMA lagging well behind JMA.



The advantage in avoiding lag is readily apparent in the chart to the right. Here we see how JMA enhances the timing of a simple crossover oscillator. The top half of the chart shows crude oil closing prices tracked by two JMA filters of different speed. The bottom half uses two EMA (exponential moving average) filters.

The oscillator becomes positive when the curve of the faster filter crosses over the slower one. This occurrence suggests a "buy" signal.

Note that JMA's crossovers are 15 and 18 days earlier! Can you afford to be 15 days late?

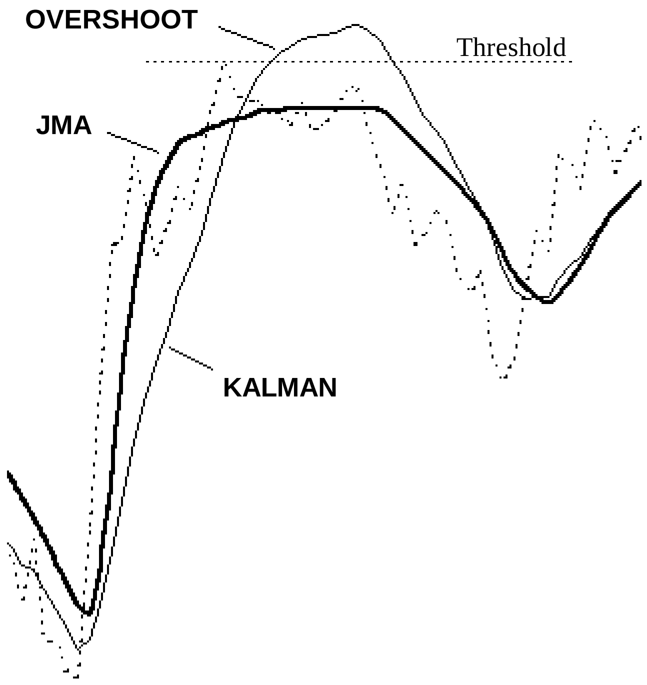


BENCHMARK #3: OVERSHOOT

Many trading systems set triggers to buy or sell when price reaches a certain threshold level. Because there is an inherent amount of noise in price action, the typical approach is to trigger when a moving average crosses the threshold. The smoothed line has less noise and is less likely to produce false alarms.

To do this right, you'll need an exceptional moving average indicator. Common versions lag too much and many sophisticated designs, like the Kalman or Butterworth filter, tend to overshoot during price reversals. Overshoots create false impressions of prices having reached levels it never truly did.

For example, in the chart to the right we see the famous Kalman filter overshoot price data, creating a false price level that the market never really achieved. DEMA filters also tend to overshoot. The overshoot crosses the shown threshold and triggers a false alarm. In contrast, **JMA did not overshoot** and thus avoided a false alarm with the user's set threshold.

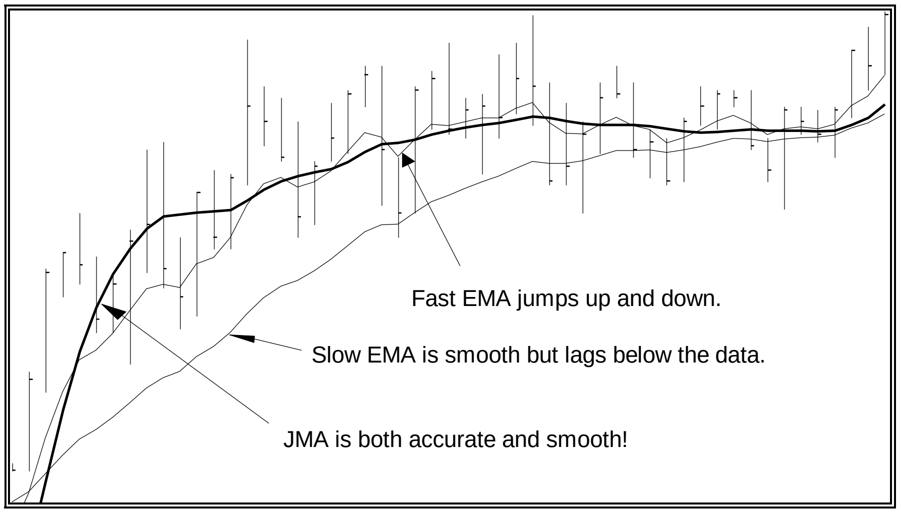


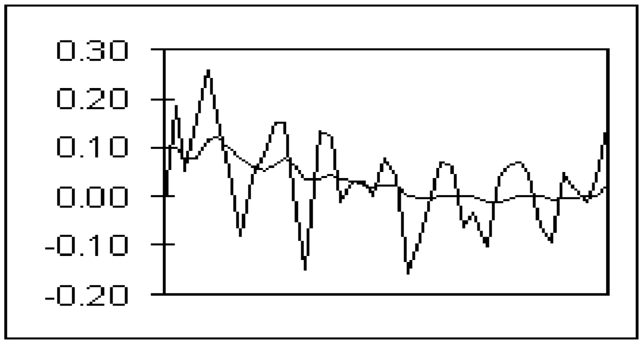
BENCHMARK #4: SMOOTHNESS

The most important property of a noise reduction filter is how well it removes noise, as measured by its smoothness.

In the chart to the right, EMA and JMA filters are run across closing prices. Note how much the fast EMA alternates upward and downward while JMA glides smoothly through the data. Clearly JMA reveals the noise-free underlying price more accurately.

If you try reducing EMA's erratic hopping by making it slower, you will discover its lag will become larger, producing late trade signals.



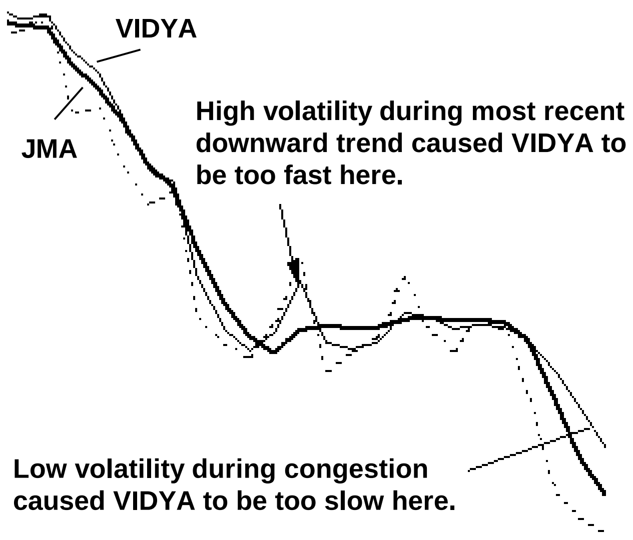


If you need a 2-bar momentum indicator, you could take the difference between two values along the EMA time series and produce the jagged line in the chart to the left. This is in contrast to the much smoother momentum signal based on JMA (flatter line). Imagine how many bad trades could be eliminated with this simple substitution!

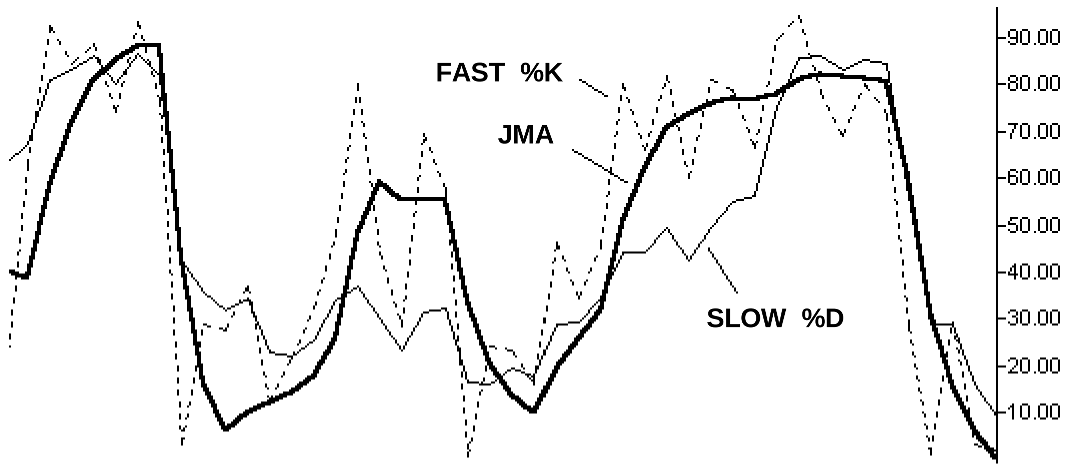
Moving averages should have consistent behavior. Some do not. For example, Chande's VIDYA is an exponential moving average whose speed is governed by the variance of price movement. Fast moving price has large variance which will eventually cause VIDYA to automatically speed up (in an attempt to prevent excessive lag). This concept sometimes works well, sometimes not.

In the chart to the right, JMA is the thick solid line and VIDYA is the thin solid line. Both perform approx-imately the same for the first 1/3 of the series. But due to the high volatility during this downward trend, VIDYA becomes hyperactive and fast tracks the choppy waves during the congestion phase of this time series. Smoothing is lost. In contrast, JMA cuts right through with a smooth horizontal line. In addition, the decrease in signal volatility soon causes VIDYA to slow down, too much in fact, as it lags behind JMA during the next downward price trend.

Comparing JMA to VIDYA



JMA also <u>enhances the accuracy</u> of technical indicators!

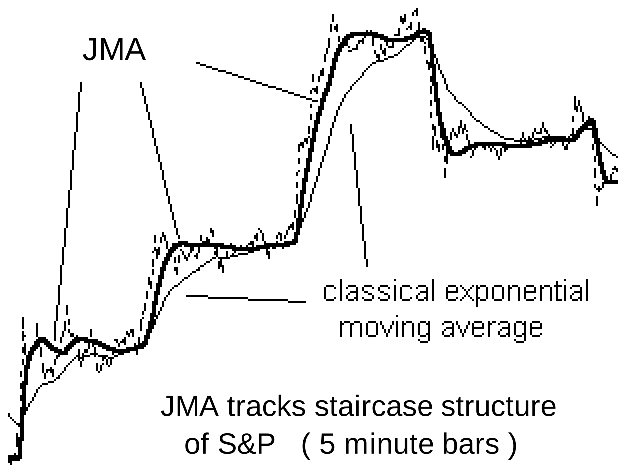


JMA resolves the riddle of how to get <u>*both smoothness and accuracy simultaneously*</u>, even with technical indicators! For example, the chart above compares the Fast %K indicator (dotted line) and two smoothed versions: one produced by the classic Slow %D (thin solid line) and the other produced by smoothing Fast %K with JMA (thick solid line). Clearly, JMA is both smoother and more accurate than slow %D!

JMA can also track price gaps produced by INTRA-DAY data. The chart on the right shows how JMA jumps to the next day's price levels while the classical exponential moving average lags behind.

Create superior trading indicators with ...

- ***better timing***
- ***less noise***
- ***greater accuracy***



JMA tracks staircase structure of S&P (5 minute bars)

Coding Applications

The DLL file contains two versions of JMA.

The **BATCH MODE** version accepts an entire array of input data and returns results into another array of equal size. This method requires ther user provide the DLL function with pointers to two arrays. This version is ideal when an entire array is available for processing with only one call to JMA.

The **REAL TIME** version accepts one input value and returns one value as a result. JMA is called for each successive value in some arbitrary time series. This approach is ideal for processing real time data, whereby the user wants an instant JMA update as each new data value arrives.

The following pages cover the following applications of JMA:

- C code example for batch mode
- C code example for real time
- Visual Basic example for batch mode
- Visual Basic example for real time

Dynamic Linking

Load Time Dynamic Linking (Microsoft Compilers)

For load-time dynamic linking, you must use the LIB file JRS_32.LIB, located at C:\JRS_DLL\LIB (or on whichever drive you specified during installation). With load-time dynamic linking, the Jurik DLL is loaded into memory when the user's EXE is loaded.

Load Time Dynamic Linking (non-Microsoft Compilers)

The LIB file we provide will only work with the MS Visual C/C++ compiler. For C/C++ users with non-Microsoft compilers, you will probably not be able to use the LIB file we have provided for Load Time Dynamic Linking with our DLL functions. You have two choices. 1) Consult your compilers' documentation to determine how to construct a LIB file from a DLL. For instance, Borland's compiler includes the IMPLIB.EXE utility to accomplish this. 2) Use run-time dynamic linking (described below). A LIB file is not required for this method.

Run Time Dynamic Linking

You may prefer to use run-time dynamic linking instead of load-time. For example, users of Microsoft Visual C may wish to prevent the Jurik DLL from automatically loading along with the user's EXE. With run-time, the DLL is loaded only when the user's EXE specifically calls for it to be loaded with the LoadLibrary function. Another reason for prefering run-time is that the user has a non-Microsoft compiler, and therefore, cannot use the LIB file provided.

For new C/C++ users, we provide sample C files which demonstrate how to accomplish run-time dynamic linking. The sample files are located in the folder C:\JRS_DLL\RUNTIME (or on whichever drive you specified during installation).

C Programming the 32 bit JMA DLL for batch mode

The file **JRS_32.DLL** contains the function JMA. In your C code, you should declare JMA as externally defined and, if using MS VC++, use the _declspec(dllimport) keywords. The function is exported as a C function, so if you are using C++, you should insert "C" (with the quotes) between the words "extern" and "_declspec". Also, you should link with JRS_32.LIB, which we provide.

```
extern _declspec(dllimport) int WINAPI JMA( int iSize, double *pdSignal,
    double dSmooth, double dPhase, double *pdFilter );
```

PARAMETERS

iSize: A 32 bit signed integer equal to the number of doubles in the input data array.
pdSignal: A pointer to an array of doubles that contain your input time series data for JMA.
dSmooth: A double precision floating point number that controls the smoothness of JMA's curve.
dPhase: A double that controls the lag/overshoot aspect of JMA's curve.
pdFilter: A pointer to an array of doubles that JMA will write its filtered time series to.

NOTES

Both input and output arrays must be of the same size, as specified by the calling parameter **iSize**.

dSmooth may be any non-negative value. Typical smoothness values range from 5 - 20.

dPhase must be between -100 to +100 inclusive. **dPhase** controls the tradeoff between lag and overshoot. Sample values of phase and its affect on JMA is shown in the table below.

Sample dPhase value	Effect on lag	Effect on overshoot
-100	maximum lag	no overshoot
0	some lag	less overshoot
+100	minimum lag	more overshoot

Although JMA reads all the input data, it does not attempt to smooth the first 30 elements of the input array. This is because JMA needs at least 30 elements to begin its statistical analysis of the data. Consequently, JMA simply copies the first 30 elements of the input array to the output array. Smoothed values begin with the 31st element.

RETURN VALUES

The JMA function returns an integer to indicate success or an error as follows:

```
#DEFINE JMASUCCESS              0      // NO ERROR
#DEFINE JMANOTINSTALLED        -1      // PASSWORD / INSTALLATION ERROR
#DEFINE JMABADDATAPTR       10111      // POINTER TO INPUT ARRAY WAS NULL
#DEFINE JMABADOUTPTR        10112      // POINTER TO OUTPUT ARRAY WAS NULL
#DEFINE JMAINSUFFICIENTDATA 10113      // JMA REQUIRES AT LEAST 31 DATA POINTS
```

The AMA function returns an integer to indicate success or error as follows:
(AMA predates JMA. AMA is no longer supported, but is still included with JMA for backward compatibility)

```
#DEFINE AMASUCCESS              0      // NO ERROR
#DEFINE AMANOTINSTALLED        -1      // PASSWORD / INSTALLATION ERROR
#DEFINE AMABADDATAPTR       11111      // INPUT POINTER WAS NULL
#DEFINE AMABADOUTPTR        11112      // OUTPUT POINTER WAS NULL
#DEFINE AMAINSUFFICIENTDATA 11113      // AMA REQUIRES AT LEAST 31 DATA POINTS
#DEFINE AMA_MEM_ERR         11114      // OUT OF MEMORY CONDITION
```

C Programming example for batch mode

```
iSize = 2500;
dSmooth = 10;
dPhase = 0;

pdSignal = (double *) GlobalAllocPtr( GHND, sizeof(double) * iLength);
pdFilter = (double *) GlobalAllocPtr( GHND, sizeof(double) * iLength);

/* At this location, put your time input series data, smoothness series data and
     phase series data into their corresponding arrays. */

error_code = JMA( iSize, pdSignal, dSmooth, dPhase, pdFilter );
```

C Programming the 32 bit JMA DLL for real time

The file **JRS_32.DLL** contains the function **JMART**. In your C code, you should declare JMART as externally defined and, if using MS VC++, use the _declspec(dllimport) keywords. The function is exported as a C function, so if you are using C++, you should insert "C" (with the quotes) between the words "extern" and "_declspec". Also, you should link with JRS_32.LIB, which we provide.

```
extern _declspec(dllimport) int WINAPI JMART( double dSeries, double
  dSmooth, double dPhase, double *pdOutput, int iDestroy, int *piSeriesID
  );
```

PARAMETERS

dSeries:	A double precision floating point number equal to the input data value
dSmooth:	A double precision floating point number that controls the smoothness of JMA's curve.
dPhase:	A double that controls the lag/overshoot aspect of JMA's curve.
pdOutput:	A pointer to the memory location of a double which contains the filter output from JMA
iDestroy:	A 32 bit signed integer, with a value = 0 or 1. When value = 1, the memory in the DLL used for a particular JMA time series is released. The desired series is designated by piSeriesID. (See next parameter) This event does not release the memory containing the output of the JMA. Control of that memory is the user's responsibility.
piSeriesID:	A pointer to the memory location of a 32 bit signed integer (iSeriesID). When processing the first element of any new time series, set iSeriesID = 0. JMA will store a unique identification number of the series into that integer (iSeriesID) pointed to by pointer piSeriesID.

NOTES

dSmooth may be any value between 2 and 500 inclusive. Typical smoothness values range from 5 - 20.

dPhase must be between -100 to +100 inclusive. **dPhase** controls the tradeoff between lag and overshoot. Sample values of phase and its affect on JMA is shown in the table below.

Sample dPhase value	Effect on lag	Effect on overshoot
-100	maximum lag	no overshoot
0	some lag	less overshoot
+100	minimum lag	more overshoot

Although JMA reads all the input data, it does not attempt to produce output for the first 30 times it is called. This is because JMA needs at least 30 input elements to begin its statistical analysis of the data. Consequently, JMA simply outputs the input value for the first 30 calls. True JMA output begins with the 31st call.

RETURN VALUES

The JMA function returns an integer, which will indicate success or an error:

0	NO ERROR
10112	POINTER TO OUTPUT MEMORY IS NULL
10114	CANNOT DEALLOCATE RAM WHEN SERIESID = 0.
10115	POINTER TO SERIES IDENTIFICATION VARIABLE IS NULL.
-1	PASSWORD / INSTALLATION ERROR. JMA OUTPUT NOT VALID.

C Programming example for real-time mode

```
// declare variables
double *pdData, *pdOutput, dSmooth, dPhase ;
int    iDestroy, iSeriesID, *piSeriesID, iErr, i ;

// get address of variable iSeriesID
piSeriesID = &iSeriesID ;

// assume you want these JMA parameter values
dSmooth = 10 ;
dPhase = -20 ;

// allocate RAM for input and output. Assume array size is 100
pdData   = (double *) GlobalAllocPtr(GHND, sizeof(double) * 100) ;
pdOutput = (double *) GlobalAllocPtr(GHND, sizeof(double) * 100) ;

// fill pdData array with double precision numbers from disk
// file or other source. (code not shown)

// clear deallocation flag and initialize series identification to 0.
iDestroy = iSeriesID = 0 ;

// loop through filtration and store results
for(i=0;i<100;i++)
    {
   iErr = JMART( *(pdData+i), dSmooth, dPhase, (pdOutput+i), iDestroy,
                    piSeriesID) ;
   if(iErr != 0)
        YourErrHandlerFunc() ;
    }

// done processing. Deallocate DLL RAM, and check for any errors
// When deallocating, it is OK to replace the output pointer with 0.
iDestroy = 1 ;
iErr = JMART( 0,0,0,0, iDestroy, piSeriesID) ;
if(iErr != 0)
     YourErrHandlerFunc() ;


// do something with data and deallocate RAM at pdData and pdOutput
```

Visual Basic example of JMA in batch mode

INTRODUCTION

In your Jurik Research DLL installation directory (e.g., C:\JRS_DLL) the workbook JMA_DMX_DLL.XLS contains a programming example using Excel's VBA to call function **JMA**. The workbook includes a worksheet where you can run the macro **JMA_Test** to run **JMA** in batch mode.

In this example, run the VBA macro called "**JMA_Test**". The macro gets the data in column 1 and sends it to the JMA batch mode function in the DLL. The output array produced by JMA is then written back onto column 5 of the worksheet.

VBA MACRO DESCRIPTION

The macro JMA_Test calls the function JMA, which is declared as shown below. Note that the input and output arrays (daInData and daOutData) are called by reference using "ByRef". This enables the calling statement to send to JMA a pointer to the first element of each data array.

```
Declare Function JMA Lib "JRS_32.dll" (   ByVal iSize As Long, _
                                          ByRef daInData As Double, _
                                          ByVal dSmooth As Double, _
                                          ByVal dPhase As Double, _
                                          ByRef daOutData As Double) As Long
```

The VBA subroutine **JMA_Test** is shown on the next page. This code will read data from column 1 of the active worksheet, call the DLL function JMA, and output its results back to the worksheet.

Note that the code calls a local subroutine "`Error_handler`". If an error condition exists, the subroutine posts a message on the screen (because JMA itself does not) and then halts the program.

```
Sub JMA_Test()

    Dim k As Long                           'iteration variable
    Dim iSize As Long                       'size of data array
    Dim iResult As Long                     'returned error code
    Dim InputData(1 To 484) As Double       'input array
    Dim OutputData(1 To 484) As Double      'output array
    Dim dSmooth As Double                   'JMA speed
    Dim dPhase As Double                    'JMA phase
    Dim calctype As Long                    'for preserving current Excel calc mode

    calctype = Application.Calculation
    Application.Calculation = xlManual

    iSize = 199         'size of input series
    dSmooth = 20        'JMA speed
    dPhase = 0          'JMA length

    ' Read Data from spreadsheet into array
    ' Input data is in column 1

    For k = 1 To iSize
        InputData(k) = Cells(k + 1, 1)
    Next k

    '--- JMA return error codes ---

    '    0      SUCCESS -- No error conditions
    '10111      did not reference first element of input array
    '10112      did not reference first element of output array
    '10113      JMA requires at least 31 data points
    '   -1      password/installation error. JMA output not valid.


   ' Call JMA using pointers to first elements of arrays
    iResult = JMA(iSize, InputData(1), dSmooth, dPhase, OutputData(1))
    If (iResult <> 0) Then
        ' Post Error Message and HALT
        Call Error_handler(iResult, calctype)
    Else
        ' Show results in column 5 on spreadsheet
        For k = 1 To iSize
            Cells(1 + k, 5).FormulaR1C1 = OutputData(k)
        Next k
    End If

    'restore calculation type
    Application.Calculation = calctype

    End Sub
```

```
' The following subroutine is a simple way to handle run-time errors that may occur
' It is good practice to handle each error type mentioned in the user manual.


Private Sub Error_handler(ByVal error_code As Long, ByVal calctype As Long)
    Dim result As Long
    result = MsgBox("Error number " & Str(error_code) &
                    " was returned by JMA.", , "JMA Error")
    Application.Calculation = calctype
    End   ' this END command will halt execution of the VBA code.
End Sub
```

Visual Basic example of JMA in real time

INTRODUCTION

In your Jurik Research DLL installation directory (e.g., C:\JRS_DLL) the workbook JMA_DMX_DLL.XLS contains a programming example using Excel's VBA to call function **JMART**. The workbook includes a worksheet where you can run the macro **JMART_Test** to run **JMART** in real-time mode.

In this example, run the VBA macro called "**JMART_Test**". The macro reads one element at a time from column 1, sequentially feeding each one through the real time version of JMA and places the results sequentially into column 4.

VBA MACRO DESCRIPTION

The function JMART is declared as shown below. Note that the output and series identification variables (dOutput and iSeriesID) are called by reference using "ByRef". The user intializes the series identification variable to zero and during the first call to JMART, the function will replace zero with an integer that uniquely identifies the time series. This way, when you have multiple time series running in parallel, the series identification numbers will tell JMART to which time series the new data point is to be assigned.

```
Declare Function JMART Lib "JRS_32.dll" ( ByVal dPrice As Double, _
                                          ByVal dSmooth As Double, _
                                          ByVal dPhase As Double, _
                                          ByRef dOutput As Double, _
                                          ByVal iDestroy As Long, _
                                          ByRef iSeriesID As Long) As Long
```

The VBA subroutine **JMART_Test** is shown on the next page. This code reads data from column 1 of the active worksheet, one element at a time, each time calling the DLL function JMART, and outputing the result back to the worksheet.

Note that the code calls a local subroutine "`Error_handler`". If an error condition exists, the subroutine posts a message on the screen (because JMA itself does not) and then halts the program.

Also note that if you have several separate data time series that you want JMA to process simultaneously in real time, **each time series must be given its own series identification variable**. In this example, only one time series will be filtered, therefore only one series identification variable needs to be declared.

```
Sub JMART_test()

    Dim dLength As Double       'JMA speed
    Dim dJMAout As Double       'JMA output
    Dim dPhase As Double        'JMA phase
    Dim iSeriesID As Long       'Input series ID code
    Dim iResult As Long         'returned error code
    Dim iDestroy As Long        'deallocate DLL RAM switch
    Dim iSize As Long           'length of data array
    Dim k As Long               'iteration variable
    Dim calctype As Long        'for preserving current Excel calc mode

    '--- JMART return error codes ---

    '    0      SUCCESS -- No error conditions
    '10112      dJMAout not declared using ByRef
    '10114      iSeriesID=0 and iDestroy=1. Cannot deallocate DLL RAM when SeriesID=0
    '10115      iSeriesID not declared using ByRef
    '   -1      password/installation error. JMA output not valid.

    iSize = 199           ' length of input array
    dLength = 20          ' JMA smoothness factor
    dPhase = 0            ' JMA phase value
    iSeriesID = 0         ' MUST initialize series identification to zero
    iDestroy = 0          ' MUST clear "deallocate DLL RAM" flag

    'disable automatic calculation
    calctype = Application.Calculation
    Application.Calculation = xlManual

    For k = 1 To iSize
        iResult = JMART(Cells(k + 1, 1), dLength, dPhase, dJMAout, iDestroy, iSeriesID)
        If (iResult <> 0) Then
            ' Post Error Message and HALT
            Call Error_handler(iResult, calctype)
        Else
            Cells(1 + k, 4).FormulaR1C1 = dJMAout
        End If
    Next k

    'deallocate DLL RAM. Check for errors.
    'iSeriesId should contain a non-zero identification value

    iDestroy = 1
    iResult = JMART(0, 0, 0, 0, iDestroy, iSeriesID)
    If (iResult <> 0) Then
        ' Post Error Message and HALT
        Call Error_handler(iResult, calctype)
    End If

    'restore calculation type
    Application.Calculation = calctype
End Sub
```

```
' The following subroutine is a simple way to handle run-time errors that may occur
' It is good practice to handle each error type mentioned in the user manual.

Private Sub Error_handler(ByVal error_code As Long, ByVal calctype As Long)
    Dim result As Long
    result = MsgBox("Error number " & Str(error_code) &
                    " was returned by JMA.", , "JMA Error")
    Application.Calculation = calctype
    End   ' this END command will halt execution of the VBA code.
End Sub
```

IF YOU FIND A BUG . . . YOU WIN

If you discover a legitimate bug in any of our preprocessing tools, please let us know! We will try to verify it on the spot. If you are the first to report it to us, you will receive the following two coupons redeemable toward your acquisition of any of our preprocessing tools:

- a $50 discount coupon
- a free upgrade coupon

You may collect as many coupons as you can.
You may apply more than one discount coupon toward the purchase of your next tool.

$$$ Anti-Piracy Reward Policy $$$

Jurik tools are world renown for excellence and value. We manage to keep costs down with large sales volume, maintained in part by protecting our copyrights with the following anti-piracy policy…

1. We have on permanent retainer one of the best intellectual property law firms in the U.S.
2. We do not perform cost-benefit analysis when it comes to litigation. We prosecute all offenders.
3. We register portions of our software with the U.S. Copyright office, entitling us to demand the offender compensate Jurik Research for all legal costs, which is typically over $10,000 per lawsuit.
4. ***We offer up to $5000 reward for information leading to the prosecution of any offender(s).***

Risk & Liability

Hypothetical or simulated performance results have certain inherent limitations. Simulated performance is subject to the fact that they are designed with the benefit of hindsight.

We must also state here that, due to the frequently unpredictable nature of the marketplace, past performance of any trading system is never a guarantee of future performance. In addition, all trading systems have risk and commodities trading leverages that risk. We advise you never to place at risk more than you can afford to lose. It's only common sense.

The user is advised to test the software thoroughly before relying upon it. The user agrees to assume the entire risk of using the software. In no event shall JRC be responsible for any special, consequential, actual or other damages, regardless of type, and any lost profit resulting from the use of this software.