AUSTRA Reference Guide

Ian Marteens

Table of Contents

Part I

1 Introduction

AUSTRA is a small functional language designed to handle financial series and common econometric models. It also implements vectors, matrices and the most frequently used operations from linear algebra, statistics, and probability.

AUSTRA expressions are efficiently parsed by a .NET Engine, and they are translated into fast-running native code that calls routines also implemented in .NET that take advantage of multicore systems and SIMD extensions.

This manual presents:

- The AUSTRA functional language, with examples.
- The output of the language is shown using the AUSTRA desktop application, which uses a variety of views to present results according to their types.

The underlying AUSTRA library is not covered by this document. Most classes and methods in the library have a direct counterpart in the language, but some operations, such as optimized fused multiply-add for vectors are currently available only from the library, though the compiler can use them via optimization.

In this section

Basic syntax
A quick tour on AUSTRA features.

Tools
A brief description of the applications for trying the AUSTRA language and library.

Reference
The reference groups language features by their functionality and data type.

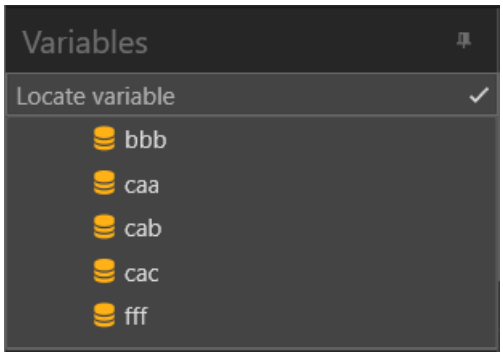
1.1 Basic syntax

This page is a quick tour on the main AUSTRA features.

Variables and definitions

The main source of data for AUSTRA is the set of series stored in a database or a file. These series can be loaded by external software, or by importing data from Excel files.

Series are sorted collections of pairs date/value and most persisted series are *raw* series representing, for instance, prices. Persisted series are listed inside the *Variables* panel, in the Austra.Desktop terminal:



The icons at the left of the series names marks them as *persistent* series. These series were loaded from some persistent store, either a database or a JSON file with the *.austra* extension. If you are using the Austra.REPL tool, you can type the show command to list existing series.

The names of these series can be directly used in AUSTRA formulas such as these:

```
aaa.rets        -- Converts a raw series into simple returns
aaa.last.value  -- Gets the last value from a series
```

AUSTRA is case-insensitive, so *AAA* and *aaa* refers to the same entity.

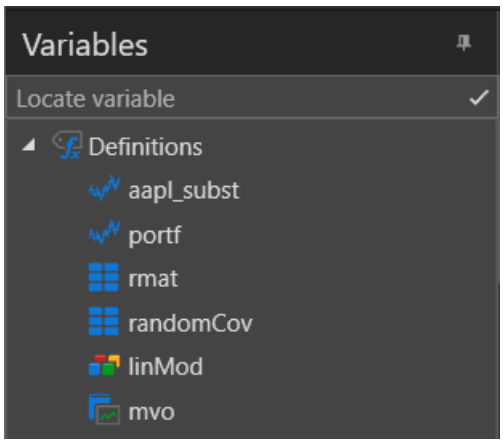
In addition to series, AUSTRA loads *definitions* from its database or persistent file. A definition is a formula with an associated name, and it can be created from AUSTRA using a command like this:

```
def cxMvo = model::mvo(sm_ret, sm_cov, sm_low, sm_high)
```

A description can also be attached to the definition using this syntax:

```
def cxMvo:"MVO model" = model::mvo(
    sm_ret, sm_cov, sm_low, sm_high)
```

Definitions are also listed inside the *Variables* panel, below their own parent node:



A definition can be removed using the **undef** command:

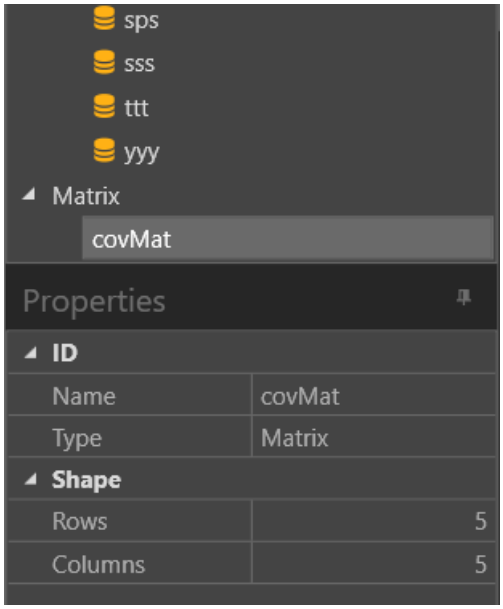
```
undef cxMvo
```

Session variables

Values can be saved just for the current session using the **set** command:

```
set covMat = matrix::cov(
    aapl.rets, msft.rets, dax.rets, esx.rets)
```

The new variable is added to the *Variables* panel and its main properties can be browsed in the *Properties* panel, when the node is selected:



Session variables are not persisted from session to session, but they can be used in expressions while the session is active:

```
covMat.evd          -- Find the eigenvectors of a covariance matrix
```

A session variable can be removed using the **set** command, without including a body:

```
set covMat
```

> **Important rule**
> When creating a definition, the parser checks that no session variables are being used in the definition's body. Session variables are not persisted, so the definition could not be able to compile the next time a session is started.

Let clauses

You can also declare temporary variables that are only valid inside the scope of a single formula. For instance, this formula has three temporary variables:

```
let m = matrix::lrandom(5),
    m1 = m * m',
    c = m1.chol in
   (c * c' - m1).aMax
```

In the above example, a lower triangular random matrix is computed, and it is multiplied by its transpose. Then, the Cholesky transform is calculated and finally we check that the transform is valid, evaluating the *absolute maximum* of the matrix difference.

The *m*, *m1* and *c* variables only exist while the formula is been evaluated.

The conditional expression

Since AUSTRA is a functional language, it doesn't have "statements". However, it provides an **if**/**then**/**else** "expression" that is completely equivalent to the function *iff()*, and its included in the language just for convenience:

```
if aapl.mean < msft.mean then aapl.mean else msft.mean
```

Of course, the above expression is just a pedantic way to write *min(aapl.mean, msft.mean)*. It can be also be expressed using *iff()* this way:

```
iff(aapl.mean < msft.mean, aapl.mean, msft.mean)
```

The **if**/**then**/**else** expression is easier to read, and it can be nested in a more readable way:

```
let c = aapl.corr(msft) in
  if c < 0 then "Negative"
  else if c < 0.5 then "Weak" else "Strong"
```

Comments

Though we do not expect anyone to write hundreds of pages of AUSTRA "script", we still support line comments for better documentation. Comments always starts with two consecutive hyphens and extends to the next line feed or the end of the expression, whatever comes first:

```
-- A cheap, longer version of MIN:
if aapl.mean < msft.mean then aapl.mean -- Another comment.
else msft.mean
```

1.2 Tools

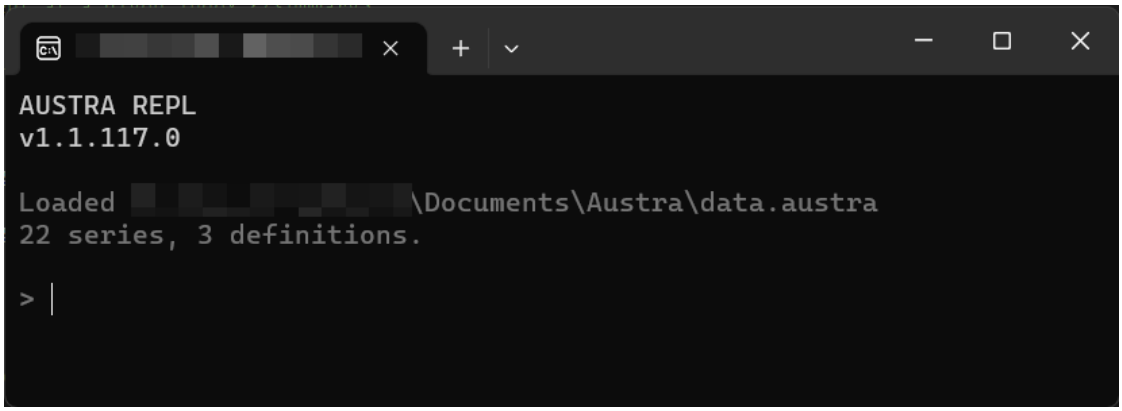
AUSTRA has currently two tools for interactively trying the language and library:

- An open source console application, that executes a simple Read-Evaluate-Print loop (Austra.REPL).
- A more sophisticated desktop application, written in WPF and using third-party components, that is not provided as part of the open source code.

We have plans to provide an open source desktop application in the near future. If possible, we will try to make it a multi-platform application.

Austra.REPL

The console application starts by looking at a file called *data.austra* in an *Austra* folder from the local *Documents* directory. If found, the application assumes it is a JSON file with series and definitions, and it creates a session with the stored data:



You can use these special commands to manage the current session:

- **import** *filename*
  Loads additional series from a CSV file into the current session.

- **export** *filename* [*series...*]
  Exports series from the current session into a CSV file. If no series are specified, all existing series are exported.

- **show**
  Displays all defined series, session variables and definitions.

- **show** *expression*
  Compiles an expression fragment to find its data type, and displays methods and properties that can be applied on the fragment. For instance, typing `show aapl.fft`, displays methods and properties for an FFT model.

- **save** [*filename*]
  Brackets indicate that the *filename* part is optional. This command saves all the content of the session in a file. If no *filename* is provided, it tries to save

the information in an *Austra\data.austra* file inside the *Documents* special folder.

- **load** [*filename*]
  Discards the current session and loads a new one from a JSON file with *.austra* extension. When no file name is provided, the command loads the default *data.austra* file, if it exists.

- **exit**
  Closes the session and terminates the application. An empty line will do the same.

Of course, most of the time you will type a formula to be evaluated:

```
> ar1.arModel(3)
Coefficients: 2,75323 -2,74031 0,985501
(R² = 0,999632)

> ar1.values.reverse[0:10]
0 0,423468 0,773358 0,994145 1,06138 0,987356 0,817725 0,619848 0,46583 0,414411
```

You can also type this command in order to show compile and execution times:

```
show @time
```

The command acts like a switch: if executed twice, times will not be shown.

Austra.Desktop

Austra.Desktop is a more sophisticated application, written in WPF and running in Windows. Some of its advantages:

- It supports more persistent formats. For instance, it can connect to a SQL database.
- Formulas are entered in a full-fledged code editor, that provides syntax highlighting and code completion. The code editor is the open-source AvalonEdit project.
- Evaluation results are shown in tabs, according to their data types. Whereas an ARModel result is shown in Austra.REPL with a succinct two-liner, a full model tab is activated in Austra.Desktop. Some models even allow for user interaction. Series use charts for their display, and some additional information, such as statistics, histogram and percentiles, are also included in the series tab. You can even drag a series from the current session catalog and drop it into an existing chart for quick comparisons.

Austra.Desktop is not currently included in the open-source project, because it was developed with third-party commercial components, mainly for the charting part. The application is now being rewritten with some enhancements in the user interface paradigm.

Part II

2 Reference

This reference groups language features by their functionality and data type.

In this section

Data types
Main data types managed by AUSTRA.

Models
Models are regular data types that the AUSTRA host displays in a different way.

Global functions
Functions that can be directly used in a formula, without requiring any prefix.

Properties
A property may refer either to a variable or a parameter-less method from an object.

Methods
Methods are also applied to objects, but they require parameters.

Class methods
Class methods are applied to a class name, instead of to an object.

Operators
AUSTRA provides a large set of binary and unary operators.

Indexers
Indexers allows access to a single element of a matrix, vector, or series, or to a range of items within them.

2.1 Data types

These are the main data types handled by AUSTRA:

double	Numeric type: 64 bits, double precision
int	Numeric type: integer, 32 bits, signed
Complex	Double precision complex numbers
string	Unicode character strings
bool	Logical type
Date	Dates: year/month/day
Series	Sorted collections of pairs date/value
Matrix	Double precision dense two-dimensional matrices
Vector	Double precision dense vectors
LMatrix	Lower triangular two-dimensional matrices
RMatrix	Upper triangular two-dimensional matrices
ComplexVector	Double precision complex dense vectors

In addition, you may find types like *Accumulator*, representing statistics, *Point*, representing a date/value pair from a series, or *EVD*, from an Eigenvalues Decomposition.

Numeric literals

Integer and real numbers are represented as in most programming languages. Here are some examples:

```
2023
1.0
-0.1E-16
```

Number literals can be suffixed with a lower-case *i* to represent an imaginary value:

```
2.0i
-3i
```

The identifier *i*, by its own, represents the imaginary unit:

```
1-3i = 1 - 3 * i
```

Complex numbers can also be created using the *complex* global function:

```
complex(1, -3) = 1 - 3 * i
complex(3) = 1 + 0i
```

Since *i* is not a keyword, you must be careful because it can be redefined as a user variable.

Date literals

Date literals come in two flavours. A simple literal only includes the month and year, assuming the first day of the month:

```
jan20
jul2021
```

Two-digit years are first interpreted as a year inside the XXI century. If the resulting date is more than 20 years ahead, 100 years are subtracted to that date. For instance:

```
jan20 = January 1st, 2020
may42 = May 1st, 1942
```

A day can be added using this syntax:

```
6@jan20
31@jul2021
```

Vector and matrix constructors

A vector is constructed by listing its components inside brackets:

```
[1, 2, 3, 4]
```

Commas are optional, so they can be omitted:

```
[1 2 3 4]
```

There's no special syntax for complex vector literals, but complex vectors can be easily created using the **complexvector**::from class method and one or two vector constructors:

```
complexvector::from([1 2 3 4], [4 3 2 1])
```

A matrix is constructed by listing its rows as vector, and then separating rows with semicolons:

```
[1, 1; 1 -1]
```

As before, commas can be omitted. New lines are always handled as whitespace.

Matrix and vector concatenation

All objects in AUSTRA are considered immutable. If you already has a matrix and you want to add new columns or rows to it, you must create a new matrix by concatenating the old matrix with the new columns or rows.

Matrix concatenation is achieved using a syntax similar to the syntax of matrix constructors. These are the allowed combinations:

```
-- Horizontal concatenation:
[matrix1, matrix2]      -- Side by side concatenation.
[matrix1, vector]       -- Vector column at the right.
[vector, matrix1]       -- Vector column at the left.
 -- Vertical concatenation:
[matrix1; matrix2]      -- Matrix upon matrix.
[matrix1; vector]       -- Vector row below.
[vector; matrix1]       -- Vector row on top.
```

Vector concatenation is a little more flexible. Though you can use only one row, you can combine vectors and scalars in the same expression:

```
-- Vector concatenation:
[1, 2, pi, vector1, 4, 5, vector2, 6, 7]
```

Commas can be omitted, but you must be careful to not create syntax ambiguity:

```
-- This expression does not compile:
[1 2 [3 4] [5 6]]
```

The above formula is invalid, since the compiler thinks the vector constructor *[5 6]* is an index on the previous constructor *[3 4]*. Of course, all of these problems are solved by using commas in case of doubt.

2.1.1 Series

The most important data type in AUSTRA is the *time series*: a sorted collection of pairs date/value.

The source of all series

Since time series contain real data most of the times, it is very unlikely to have them created in code. Series in AUSTRA are loaded from an external persistent source, such a file or a database.

- For files, the most complete format is a proprietary JSON format, that allow both series and definitions to be saved and retrieved. This is file format used by Austra.REPL, the console application.
- Series can be imported from an Excel file, either in XLSX or CSV format.
- Currently, AUSTRA supports a connector for SQL Server databases.

In any case, it's very easy to create an adapter to load series from any other service or format. Series available to the parser are stored in an instance of a class that should implement the *IDataSource* interface, defined at the *Austra.Parser* library. It's easy to add any series to that object using an *Add* method from the interface.

Additional information

Since one of the main goals of AUSTRA is to deal with financial time series, there is a number of optional properties that can be stored in a series:

- **Name**: the name of the series is the name that is used by the parser to locate a series. For this reason, the series' name must be a valid identifier.
- **Ticker**: however, it's frequent for series to be identified by traders by their *tickers*, which is the name assigned by the provider of the series. A ticker is not necessarily a valid identifier, so we provide two different fields, one for the name and the second for a ticker. Tickers can be empty.
- **Frequency**: each series has an associated frequency, which can be daily, weekly, biweekly, monthly, bimonthly, quarterly, semestral, yearly, or undefined. The library, at run time, checks that both operands in a binary operation have always the same frequency.
- **Type**: in addition, each series has a type that can be either *Raw*, *Rets*, *Logs*, *MixedRets* or *Mixed*.

Series versus vectors

Vector operations check, at run time, that the operands have the same length. The same behaviour would be hard to enforce for series. On one hand, each series can have a different first available date. On the other hand, even series with the same frequency could have reported values at different days of the week or the month, and still, it could be interesting to mix them.

So, the rules for mixing two series in an operation are:

- They must have the same frequency, and their frequencies are checked at runtime.
- However, they may have different lengths. If this is the case, the shorter length is chosen for the result.
- The points of the series are aligned according to their most recent points.
- The list of dates assigned to the result series is chosen arbitrarily from the first operand.

2.2 Models

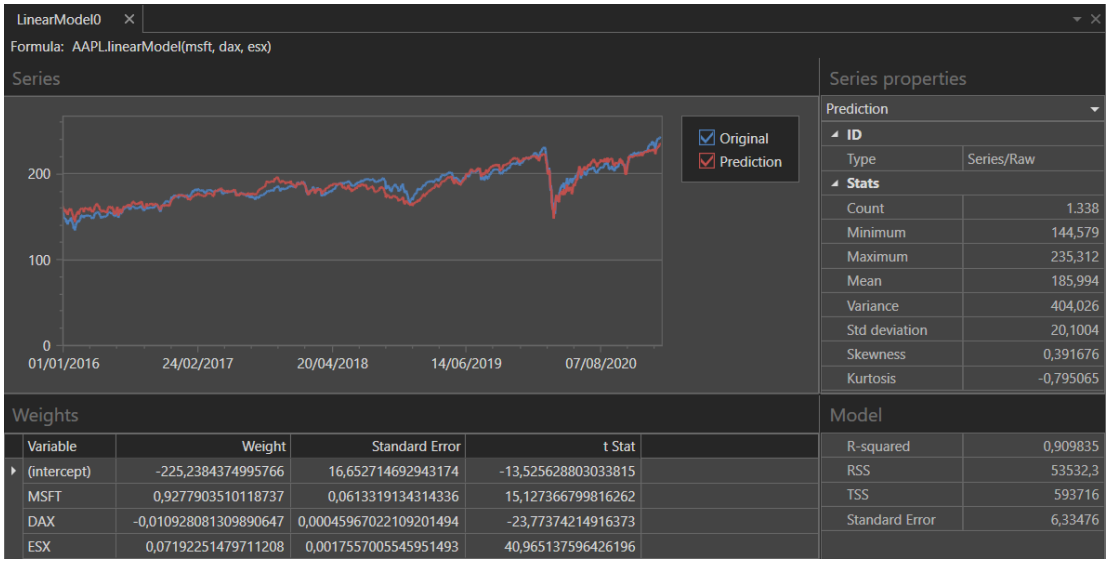
Models are regular data types that have properties and methods like any other type. The reason to handle them in a separate topic is that the AUSTRA hosting usually has special provisions for their display. Some models only contain information, but other models are designed for visual interaction.

Linear model

Linear models are returned by the linearModel() method from series and vectors.

```
set lm = aapl.linearModel(msft, dax, es)
```

The series the method is applied on, that is, the one at the left of the method name, is the series to be explained by a linear combination of the series passed as arguments. The **set** clause in the above example is optional, but it is included for convenience. You can also create a linear model on the fly for displaying. The exact visualization of this model depends on the host, but this is how it looks in the desktop terminal:



These are the properties that can be extracted from the linear model:

original	The original series.
prediction	It is the predicted linear combination of the remaining series.
weights	The weights of the linear combination.
r2	The R^2 statistics: explained variance over total variance.
rss	Residual sum of squares.
tss	Total sum of squares.

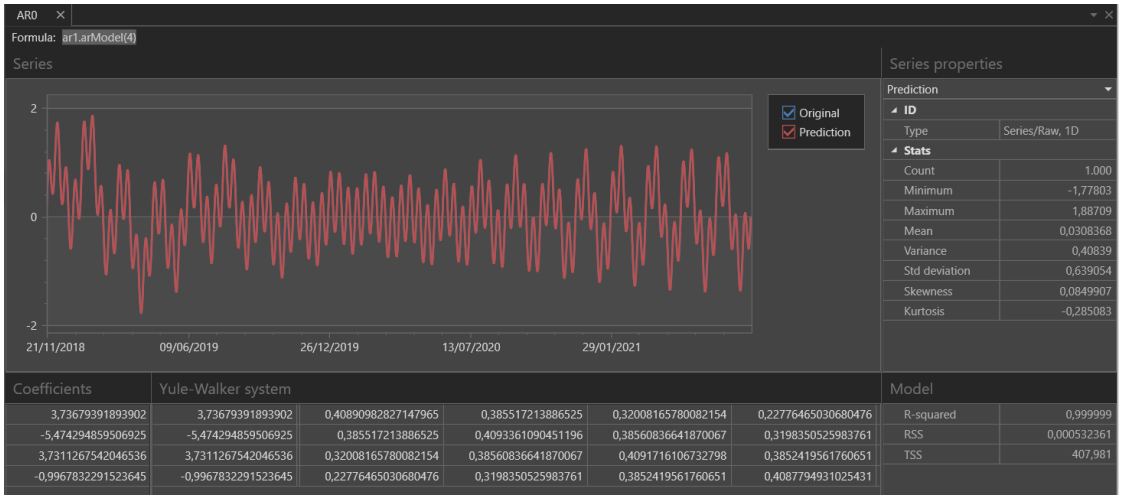
When *linearModel()* is called on a vector, properties *original* and *prediction* return vectors instead of series.

Autoregressive model (AR)

Autoregressive (AR) models are created by the arModel(n) method call, available both for series and vectors:

```
aapl.arModel(4)
```

Coefficients are calculated using the least squares method.



These are the properties that can be extracted from an AR model:

original	The original series.
prediction	The series predicted by the model.
coefficients	The model's coefficients.
r2	The R^2 statistics: explained variance over total variance.
rss	Residual sum of squares.
tss	Total sum of squares.

When *arModel()* is called on a vector, properties *original* and *prediction* return vectors instead of series.

Mean Variance Optimizer (MVO)

The *Mean Variance Optimizer* (MVO) model is created using a class method.

There are two variants for this method call. The simplest one is this:

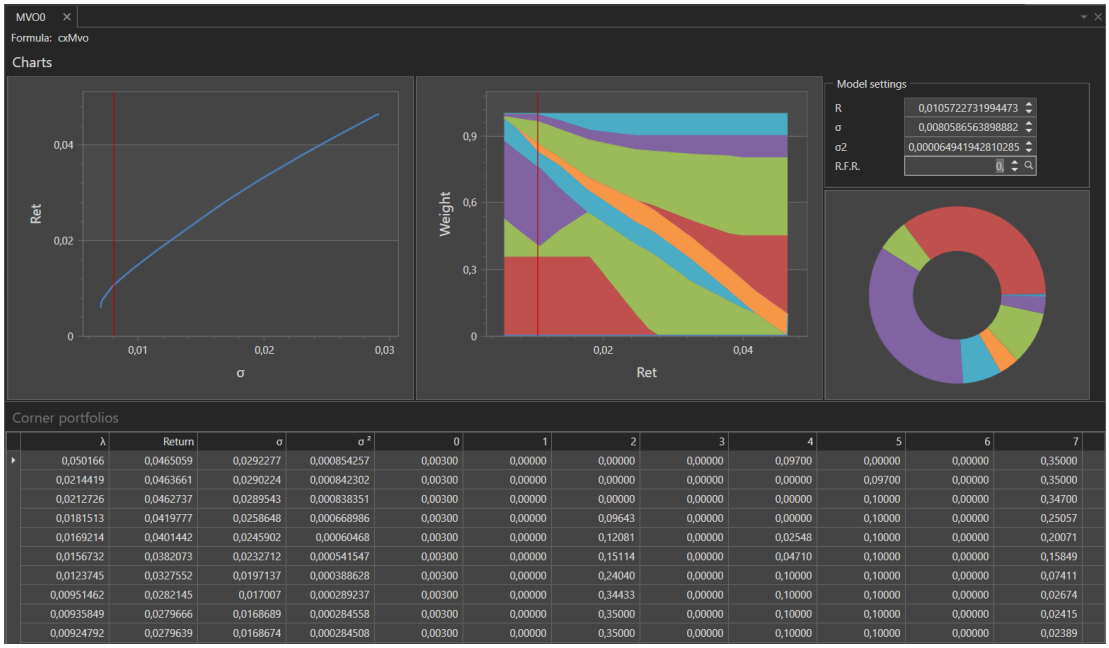
```
model::mvo(expReturns, covariances)
```

You must supply a vector with expected returns, in the first parameter, and a covariance matrix, in the second one. In this case, weights are allowed to float freely between 0.0 and 1.0. Weights can be restricted passing another two vector parameters:

```
model::mvo(expReturns, covariances, lowLimits, highLimits)
```

In both cases, you can add a list of series or character strings after the last parameters, to give names to the considered assets.

The desktop terminal shows a view like this for MVO models:



λ	Return	σ	σ²	0	1	2	3	4	5	6	7
0,050166	0,0465059	0,0292277	0,000854257	0,00300	0,00000	0,00000	0,00000	0,09700	0,00000	0,00000	0,35000
0,0214419	0,0463661	0,0290224	0,000842302	0,00300	0,00000	0,00000	0,00000	0,00000	0,09700	0,00000	0,35000
0,0212726	0,0462737	0,0289543	0,000838351	0,00300	0,00000	0,00000	0,00000	0,00000	0,10000	0,00000	0,34700
0,0181513	0,0419777	0,0258648	0,000668986	0,00300	0,00000	0,09643	0,00000	0,00000	0,10000	0,00000	0,25057
0,0169214	0,0401442	0,0245902	0,00060468	0,00300	0,00000	0,12081	0,00000	0,02548	0,10000	0,00000	0,20071
0,0156732	0,0382073	0,0232712	0,000541547	0,00300	0,00000	0,15114	0,00000	0,04710	0,10000	0,00000	0,15849
0,0123745	0,0327552	0,0197137	0,000388628	0,00300	0,00000	0,24040	0,00000	0,10000	0,10000	0,00000	0,07411
0,00951462	0,0282145	0,017007	0,000289237	0,00300	0,00000	0,34433	0,00000	0,10000	0,10000	0,00000	0,02674
0,00935849	0,0279666	0,0168689	0,000284558	0,00300	0,00000	0,35000	0,00000	0,10000	0,10000	0,00000	0,02415
0,00924792	0,0279639	0,0168674	0,000284508	0,00300	0,00000	0,35000	0,00000	0,10000	0,10000	0,00000	0,02389

You can interact with the model using the editors in the upper right panel. For instance, changing *R* locates a portfolio with the specified return. R.F.R. stands for *Risk Free Return*, and you can click the button inside the editor to find the portfolio with best Sharpe ratio for that risk free return.

Rows in the lower grid can also be double clicked, to find the associated point in the charts.

2.3 Global functions

These functions can be directly used in a formula, without requiring any prefix. Most of them takes a numeric parameter and returns a real value.

abs	Absolute value.
exp	Exponential.
log	Natural logarithm.
sin	Sine.
cos	Cosine.
tan	Tangent.
pi	The π constant.
asin	The angle whose sine is the specified parameter.
acos	The angle whose cosine is the specified parameter.
atan	The angle whose tangent is the specified parameter.
round(x)	Rounds to the nearer integer value.
round(x, digits)	Rounds with a given number of digits.
truncate	Calculates the integer part of a number.

sqrt	The square root of the parameter.
erf	The Error function.
ncdf	The cumulative of the standard normal distribution.
probit	Inverse of the cumulative of the standard normal distribution.
gamma	The Gamma function (Γ).
lnGamma	The logarithm of the Gamma (Γ) function.
beta	The Beta function.
min	The minimum of two numeric values, or dates.
max	The maximum of two numeric values, or dates.
today	The current date.
iff	A conditional function with three arguments
polyEval	Evaluates a polynomial given a complex or real argument.
polyDerivative	Evaluates the derivative of a polynomial given an argument.
polySolve	A solver for polynomial equations.
solve	A Newton-Raphson solver. See below for details.
random	Generates a random number from a uniform distribution.
nrandom	Generates a random number from a standard normal distribution.

The *beta* function needs two real arguments:

```
beta(1, 2) = beta(2, 1)
```

min() and *max()* are functions with two arguments. If more arguments are needed, you can always combine them or put the arguments in a vector and then call the max or min method for the vector:

```
max(1, max(2, 3)) = [1, 2, 3].max
```

today is a parameter-less function, and returns the current day, with no time fraction:

```
today > jan2000
```

iff(), on the other hand, is a shortcut for the **if**/**then**/**else** expression, taking three arguments. *iff()* is a "lazy funcion", in that it doesn't evaluates all of its arguments, but just those needed.

The Newton-Raphson solver is a function accepting from three up to five arguments:

```
solve(x => sin(x) - 1, x => cos(x), 0, 1e-9, 100)
```

The first two arguments are lambda functions: one for the function we want to solve for a root, and the second for the first derivative of that function. Please note that the function does not verify that the lambda function and its derivative lambda match. The third argument is required, and represents the initial guess to start running the algorithm. Again, a bad guess may make the algorithm fail.

The fourth and fifth arguments can be omitted. The fourth parameter is the desired accuracy, and when omitted, it defaults to 1e-9. The last parameter is the maximum numbers of iterations, which by default is 100.

Polynomials

The *polySolve* function takes either a vector or a list of reals and considers them as the coefficients of a polynomial. The first value is the coefficient of the highest degree term. For instance, the vector *[1, 2, 3, 4]* stands for the polynomial:

```
x^3 + 2*x^2 + 3*x + 4
```

This function can throw an exception if it does not know how to handle a given polynomial, or when there are no available roots. The returned value is always a complex vector, even when all roots are real-valued. For instance:

```
> polySolve(1, 2, 3)
<-1; 1,41421> <-1; -1,41421>
```

The inverse of *polySolve* is *polyEval*, which takes either a complex or a real, and a list of coefficients, either in a single vector or as a list of real values, and evaluates the polynomial at the supplied value:

```
> polySolve(1, 2, 3)
<-1; 1,41421> <-1; -1,41421>
> polyEval(ans[0], 1, 2, 3)
<-4,440892098500626E-16; 0>
```

If you have a *poly* variable containing polynomial coefficients, you can check the answers from the solver like this:

```
polySolve(poly).all(c => abs(polyEval(c, poly)) <= 1e-14)
```

Of course, the accuracy of the roots may vary according to the polynomial.

A close relative of *polyEval* is *polyDerivative*, which calculates the derivative of a given polynomial at the specified argument:

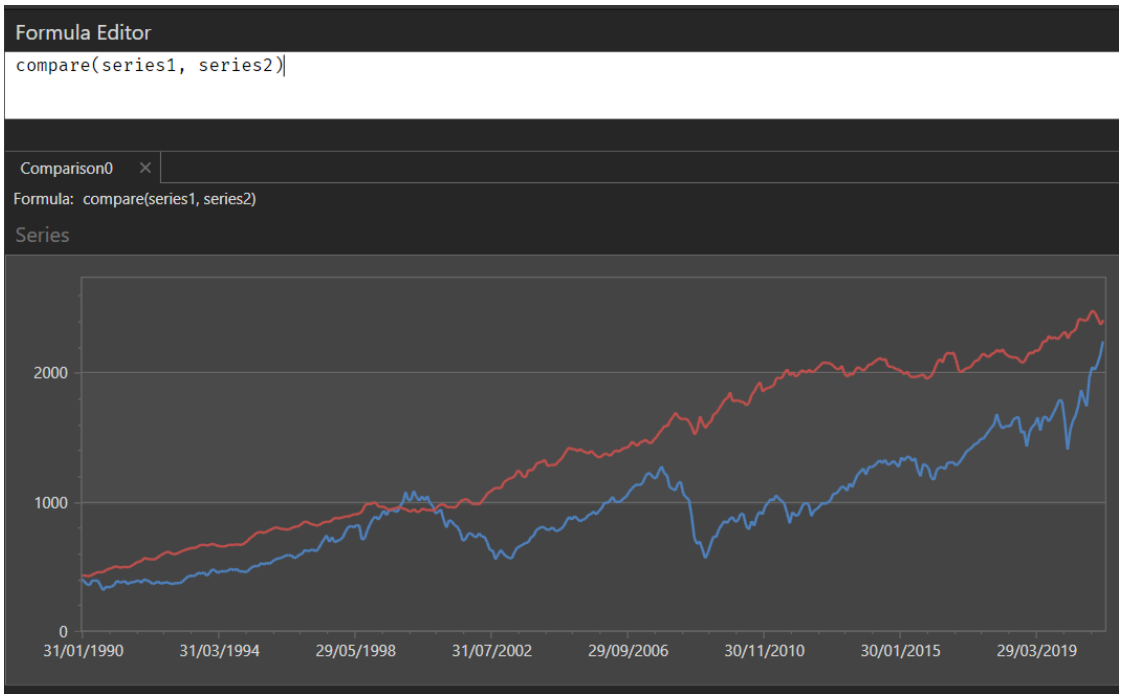
```
set v = [1, 2, 3, 4]
polyEval(2, v) = 26
polyDerivative(2, v) = 23
```

polyDerivative can be useful when finding a real root for a polynomial using the Newton-Raphson algorithm.

Compare

compare() is a special global function that groups two arguments of the same type, most of the times, series or vectors. It's also syntactically available as a method of the class `model`.

The AUSTRA terminal should then include both series in a same chart, for comparison purposes:



When vectors are compared, values are shown, side by side, in a grid:

Vector

Item	First vector	Second vector
0	2256,4439688121083	2420,989839187276
1	2149,289099526066	2385,335947929211
2	2084,4748509402225	2432,1219610479684
3	2034,6200886714562	2474,719575876734
4	2056,291086989756	2496,708108326922
5	1974,575752942974	2463,6156060583194
6	1752,7365846200878	2419,621560561014
7	1809,5627579880745	2417,309518231726
8	1877,014218009478	2426,0357576222045
9	1761,9477144167552	2429,7402088997073
10	1683,0683381745903	2354,6426252636143
11	1641,8590429597916	2333,9075289815073
12	1569,2172450695607	2323,71336167226
13	1416,2436936248275	2278,9694887631754
14	1636,6916373643169	2331,2035923494004
15	1790,5595474698052	2315,628810102589
16	1802,8359578046163	2286,4427388433865

Vector properties

Second vector

Shape	
Length	580
Norm	29942,9
Stats	
Minimum	100
Maximum	2496,71
Mean	989,497
Variance	567693
Std deviation	753,454

A quicker way to compare two series is to show first one of them in its proper window, and then to drag a series from the list at the right of the application and drop it on the chart of the first series. Then, the chart would show both series at the same time, and the percentiles chart will also show percentiles for both series.

2.4 Properties

An AUSTRA property may refer either to a variable or a parameter-less method from an object. Since parameter-less methods don't require an empty set of parentheses, both features are indistinguishable in AUSTRA.

Dates

Dates support these properties:

day	Day of month.
month	Month: 1 is January.
year	The year.
dow	The day of the week.
isleap	Does the date belong to a leap year?

Complex numbers

Complex numbers support these properties:

real	Gets the real part of the complex number.
imaginary	Gets the imaginary part of the complex number.
magnitude	Gets the magnitude of the complex.
phase	Gets the phase, in radians.

Series

These are the properties supported by series:

length	Number of points.
min	Minimum value.
max	Maximum value.
amax	Maximum absolute value.
amin	Minimum absolute value.
mean	Mean value.
var	Variance.
varp	Population's variance.
std	Standard deviation.
stdp	Population's standard deviation.
skew	Skewness.
kurt	Kurtosis.

sum	The sum of all values.
first	First point in the series.
last	Last point in the series.
stats	Gets all statistics in a single object.
rets	Converts a raw series into a linear returns one.
logs	Converts a raw series into a logarithmic returns one.
movingRet	Monthly/yearly moving return.
ncdf	Percentile of the last value, according to the normal distribution.
fft	Fast Fourier Transform.
perc	Percentiles.
values	Gets all values from the series as a vector.
random	Generates a random series using the mean and variance.
fit	Fits the series to a line segment and returns the two coefficients in a vector.
linearFit	Fits the series to a line segment and returns the corresponding series.
spline	Creates a cubic interpolator for the series.

Spline interpolation can be easily applied to series by creating a cubic piecewise interpolator via the *spline* property. This example creates a spline and uses it immediately to interpolate a value from a given date:

```
msft.spline[4@jul2020]
```

You must pass a valid date to the indexer of the spline in order to retrieve an interpolated value. You can also retrieve any of the generated cubic polynomials for a given segment, for closer examination. In that case, you must pass an index to the *poly* method of the spline. Note that you can also supply an index relative to the end:

```
> ar1.spline.poly(^2)
-0,00150689x³ + 0,0740303x² + 0,135476x + -0,53293
```

More general splines can also be generated using the spline::new class method, which takes two vectors as arguments.

Matrices

These are the properties for matrices:

rows	Number of rows.
cols	Number of columns.
det	The determinant of the matrix.
trace	The sum of values in the main diagonal.
diag	Gets the main diagonal as a vector.
chol	The Cholesky Decomposition.
evd	Eigenvalues Decomposition.

amax	The maximum absolute values of all cells.
amin	Minimum absolute value.
min	Minimum value.
max	Maximum value.
inverse	Return the inverse of a square matrix.

The matrix inverse and the determinant are computed using the LU factorization, for more numerical accuracy.

Vectors

These are the properties supported by real vectors:

length	Number of components.
norm	The square root of the sum of squares.
first	A shortcut for getting the first value.
last	A shortcut for getting the last value.
sqr	The inner product with itself.
sqrt	Point-wise square root.
sum	The sum of all components.
fft	Fast Fourier Transform.
amax	The maximum absolute values of all components.
amin	Minimum absolute value.
min	Minimum value.
max	Maximum value.
distinct	Creates a new vector with all distinct values in the original.
sort	Creates a new vector with values sorted in ascending order.
stats	Statistics on the vector's components.

Complex vectors support a subset of the above properties and some additional ones:

length	Number of components.
norm	The square root of the sum of squares.
amax	Maximum absolute value.
first	A shortcut for getting the first value.
last	A shortcut for getting the last value.
sqr	The inner product with itself.
sum	The sum of all components.
fft	Fast Fourier Transform.
real	The real parts of the vector items as a Vector.
distinct	Creates a new vector with all distinct values in the original.
imaginary	The imaginary parts of the vector items as a Vector.
magnitudes	The magnitudes of the vector items as a Vector.
phases	The phases of the vector items as a Vector.

The *evd* property of a matrix returns an object with these properties:

values	A complex vector with all eigenvalues.
vectors	A matrix formed by eigenvectors in columns.
d	A block diagonal matrix with eigenvalues as real numbers.
rank	The rank of the original matrix.
det	The determinant of the original matrix.

For a symmetrical matrix, the *d* property of a EVD decomposition satisfies this formula:

```
let lm = matrix::lrandom(32), m = lm * lm', e = m.evd in
   (m - e.vectors * e.d * e.vectors').amax < 1e-12
```

For a non-symmetrical matrix, the corresponding test is:

```
let m = matrix::random(32), e = m.evd in
   (m * e.vectors - e.vectors * e.d).amax < 1e-12
```

Both series and real vectors support a *stats* property returning an object with these properties:

length	Number of samples.
min	Minimum value.
max	Maximum value.
mean	Mean value.
var	Variance.
varp	Population's variance.
std	Standard deviation.
stdp	Population's standard deviation.
skew	Skewness.
kurt	Kurtosis.

The *fft* property, both for complex and real vectors, returns a value from the *FftModel* class, with these nested properties:

length	Number of values in the *amplitudes* and *phases*.
magnitudes	Amplitudes from the spectrum.
phases	Phases from the spectrum.
values	The full spectrum as a complex vector.
inverse	Calculates the inverse FFT of values.

The length of the *values* vector may be the double of the length of *magnitudes* and *samples*. The full spectrum of a real vector's FFT has a redundant second half, so only the first half is included in those real vector properties.

The return type of *inverse* depends on the type of the original vector: it return a real vector for a real original sample, and a complex vector, otherwise.

2.5 Methods

Methods are somehow like properties: they are always applied to an object. On the other hand, they require arguments, that must be passed in a parentheses-enclosed list.

These are the methods supported by dates:

addMonths(months)	Adds a number of months to the date
addYears(years)	Adds a number of years to the date

All of them takes a single integer parameter.

Series support these methods:

cov(series)	Covariance of the series and the parameter
corr(series)	Correlation between the series and the parameter
autocorr(lag)	Autocorrelation, given a lag.
correlogram(size)	Autocorrelation values, from 1 to size.
linear(series...)	Calculates a linear regression using the series in the parameters as predictors.
linearModel(series...)	Like the above method, but returning more information about the regression.
ar(degrees)	Returns the coefficients of an autoregressive model.
arModel(degrees)	Like the above method, but returning more information about the model.
stats(date)	Gets statistics for the month and year from the passed date.
ncdf(value)	Percentile of an arbitrary value, according to the normal distribution.
movingAvg(points)	Simple Moving Average of the series.
movingStd(points)	Simple Moving Standard Deviation of the series.
movingNCdf(points)	Moving Percentile according to the Normal Distribution.
ewma(alpha)	Exponentially Weighted Moving Average.
indexof(value)	Returns the index of the first occurrence of a value.
map(*function*)	Transforms a series using a lambda function.
all(*predicate*)	Checks if all points in the series satisfy the predicate.
any(*predicate*)	Checks if any point in the series satisfies the predicate.
filter(*predicate*)	Creates a new series with points satisfying a predicate.
zip(*series, zipper*)	Combines two series into a new one.

Both *autocorr* and *correlogram* implement an efficient algorithm that does not assume the series to be stationary, so the results are more accurate.

Real and complex vector methods are these:

indexof(value)	Returns the index of the first occurrence of a value.
map(*function*)	Transforms a vector using a lambda function.
mapreal(*function*)	Transforms a complex vector into a real vector.
all(*predicate*)	Checks if all items in the vector satisfy the predicate.
any(*predicate*)	Checks if any item in the vector satisfies the predicate.
filter(*predicate*)	Creates a new vector with items satisfying a predicate.
zip(*vector, zipper*)	Combines two vectors into a new one.

Autocorrelation, and autoregressive and linear models can also be calculated on values from a real vector:

autocorr(lag)	Autocorrelation, given a lag.
correlogram(size)	Autocorrelation values, from 1 to size.
linear(vectors...)	Calculates a linear regression using vectors in the parameters as predictors.
ar(degrees)	Returns the coefficients of an autoregressive model.
arModel(degrees)	Like the above method, but returning more information about the model.

These are the methods supported by matrices:

getRow(row)	Extracts a row given its index
getCol(col)	Extracts a column given its index
map(*function*)	Transforms a matrix using a lambda function.
all(*predicate*)	Checks if all cells in the matrix satisfy the predicate.
any(*predicate*)	Checks if any cell in the matrix satisfies the predicate.

Lambdas

Methods such as *map*, *any* and *all* are special in that they accept an anonymous function (*lambda*) as an argument. Let's start with a simple expression:

```
prices.all(x => x >= 0)
```

The above formula checks whether all values in the *prices* series are non-negative. That's the role of the *all* method, which checks that all values in a series satisfies a given predicate. The way we state the predicate to be satisfied is using this syntax:

```
x => x >= 0
```

This can be read as *"given an arbitrary value x, check that it is non-negative"*. We can use *all* for any other purpose, such as checking that all values in a series lie inside the [0, 1] interval:

```
prices.all(value => 0 <= value <= 1)
```

Notice that in this new example, we have used another value for the "arbitrary given value".

This example shows how to use the related method *any*:

```
prices.any(x => x >= 1)
```

In this case, we are checking whether exists at least one value in *prices* that is above 1.

Both *any* and *all* require a predicate as argument: a formula that given an arbitrary value, returns true or false. The *map* method, instead, requires a more general function that converts a real value into another one. Let's say we want to limit values from a series, so that no one is greater than 1000:

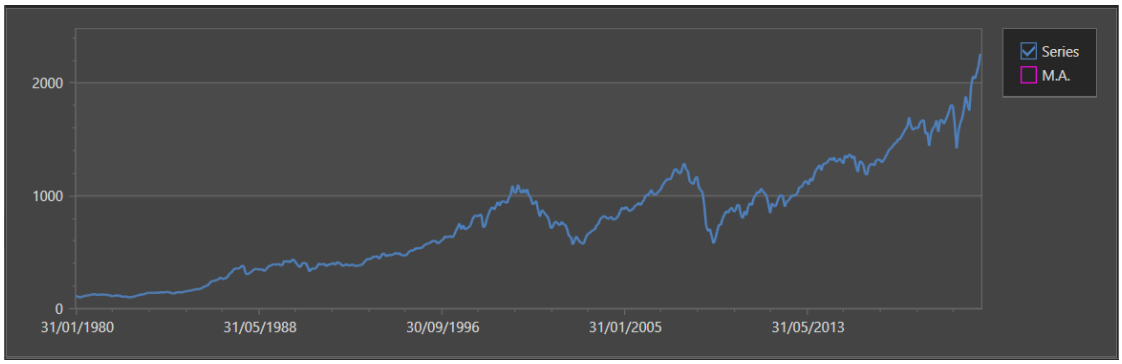
```
prices.map(x => min(x, 1000))
```

The *ncdf()* method of a series takes a real value and classifies it according to its position in the normal distribution implicitly defined by the series. By definition, it is a value between 0 and 1. Even better, *ncdf()* is monotonic: if $x < y$, then $s.ncdf(x) < s.ncdf(y)$. All this means that this method is a nice way to compress an arbitrary series so all their values lie between 0 and 1, while preserving the shape of the series.

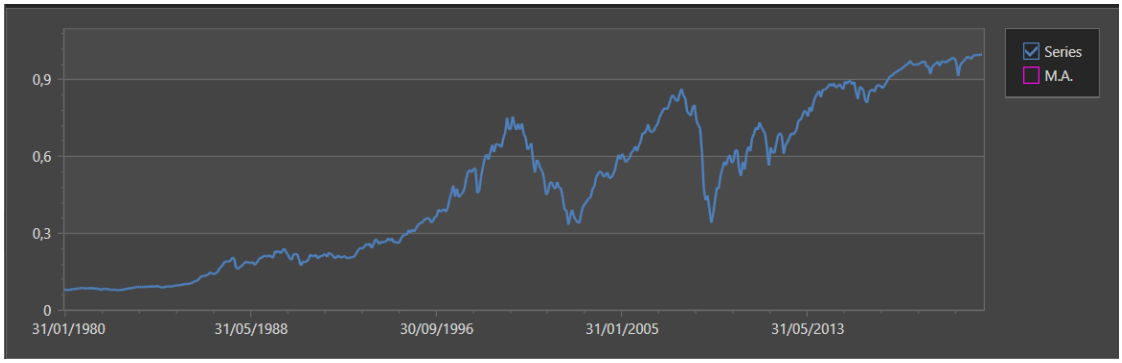
This formula does the trick:

```
prices.map(x => prices.ncdf(x))
```

Nothing remarkable here: *prices* is a global identifier, and it should not surprise us that we can use it both in the main formula and in the nested lambda. This is the original series:



And this is the compressed one:



Please note that the main difference between both charts is the range of values.

What if what we really wanted was the compressed series with the simple returns of *prices*? Not a big deal. This, obviously, works:

```
prices.rets.map(x => prices.rets.ncdf(x))
```

But we can do it better, using a let clause:

```
let pr = prices.rets in
    pr.map(x => pr.ncdf(x))
```

Though *pr* is a local variable defined in the main body of the formula, we still can reference it from our nested lambda function.

Complex vectors also accept a *map* method, but the lambda's signature is different: it receives a complex and must return a complex.

```
aapl.fft.values.map(c => c')
```

There's a related *mapreal* method that only works with complex vectors. Its lambda must take a complex and convert it to a real value:

```
aapl.fft.values.mapreal(c => c.real)
```

Finally, the *filter* method takes a predicate, like *all* and *any*, but creates a new vector with those items satisfying the predicate:

```
prices.rets.filter(x => x > 0)
-- Conditions like this always hold:
prices.rets.filter(x => x > 0).all(x => x > 0)
```

filter can return an empty vector if no item satisfies the predicate.

Filters can also be applied to a series, but in this case, the predicate accepts a point containing a date and a value:

```
aapl.filter(p => p.date.year = 2020)
```

Zip

zip() is just another method that takes a lambda, but this time, the lambda requires two parameters. Let's say we have two series, and we want to create the one with the maximum value at each point. The *zip* method was created for this task:

```
series1.zip(series2, (x, y) => max(x, y))
```

This lambda function requires two parameters: *x* will be assigned values from the first series and *y* will take values from the second series.

As a rule, when a lambda function has two parameters, they must be enclosed inside parentheses.

2.6 Class methods

Class methods are applied to a class name, instead of to an object. For instance:

```
matrix::random(5)
```

Please note that the method is separated from the class name by two consecutive colons.

Class methods can be considered as named constructors in the AUSTRA formula language. They avoid the pollution of the global name space: you can have, for instance, a class method **vector**`::new` and also a **matrix**`::new` method. Internally, they can be implemented in C# either by constructors or by static methods.

Series

The *Series* class supports this class method:

new(weights, series...)	The first parameter is a vector of weights, and from that point on, a list of series must be included.

This class method creates a linear combination of series. The length of the weights vector can be equal to the number of series or the number of series plus one. For instance:

```
series::new([0.1, 0.9], aapl, msft)
```

The above call returns this series:

```
0.1 * aapl + 0.9 * msft
```

We can add another item to the vector:

```
series::new([0.5, 0.1, 0.9], aapl, msft)
```

In this case, the result is equivalent to this:

```
0.5 + 0.1 * aapl + 0.9 * msft
```

Vector

The *Vector* class supports these class methods:

zero(size)	Creates a vector with *size* zeros
ones(size)	Creates a vector with *size* ones
random(size)	Creates a vector with *size* items, using a uniform distribution
nrandom(size)	Creates a vector with *size* items, using a standard normal distribution
new(length, lambda)	Initializes a vector using an arbitrary lambda function.

Random vectors created by using *random* and *nrandom* are easily scaled and shifted using vector operators.

vector::new is a very flexible technique to create and initialize a vector, using an arbitrary function on the item index. For instance:

```
vector::new(1024, i => sin(i*pi/512) + 0.8*cos(i*pi/256))
```

In this case, you could perform a Fast Fourier Transform (FFT) on the result, to check the harmonic content of the vector.

vector::new can also be used to create a linear combination of vector, as with series:

```
vector::new([0.5, 0.1, 0.7, 0.2], v1, v2, v3)
```

The above expression is equivalent to this one:

```
0.5 + 0.1 * v1 + 0.7 * v2 + 0.2 * v3
```

Please note that the parser can detect some code patterns and optimize expressions automatically. For instance, for vectors, the parser recognizes these patterns:

```
vector1 * scalar + vector2
scalar * vector1 + vector2
```

Both these expressions are reduced to calls to either the method `Vector.MultiplyAdd(double, Vector)` or `Vector.MultiplyAdd(Vector, double)`. Both methods are internally optimized to use a single temporary buffer, instead of the two buffers of a naïve implementation, and both of them use FMA fused operations when available. Of course, the method underlying the above presented **vector**`::new` constructor is even better optimized and runs several faster than even the optimized versions of lineal composition.

There's a third variant of **vector**::new and **complexvector**::new, that differs in the number of parameters of the lambda:

```
vector::new(1024, (i, v) => if i = 0 then 1 else 0.8 * v[i - 1])
```

In this case, the additional parameter *v* represents the vector or complex vectors being initialized. Of course, only entries with index lesser than *i* have values different from zero.

This class method can be used, together with safe indexers, to create autoregressive series:

```
set arData = let r = vector::nrandom(1024) in
    vector::new(r.length,
        (i, v) => r[i] + 0.8 * v{i - 1} + 0.2 * v{i - 2})
```

Safe indexers are the key to avoid out of range exceptions for the first few entries in the built vector.

Spline

Temporal series have their own spline property for interpolation. You can also create an interpolator from two vectors using **spline**::new:

```
set sp = spline::new(
    vector::new(1024, i => i * pi / 512),
    vector::new(1024, i => sin(i * pi / 512)))
```

In this case, the object returned by the method requires a double value in its indexer:

```
sp[pi/3]
```

The **spline**::grid class method simplifies approximating a function with a piecewise cubic spline:

```
set sp = spline::grid(0, 2pi, 1024, x => sin(x))
```

You can even compute the inferred derivative at any point of the supported interval:

```
> sp[pi/4]                    -- sin(π/4)
0.7071067811865476

> sp.derivative(pi/4)         -- cos(π/4)
0.7071067811809789
```

This spline also provides a *poly* method receiving an index for retrieving a given polynomial:

```
sp[1]       -- An interpolated value for argument 1.0.
sp.poly(1)  -- The second cubic polynomial in the spline.
sp.poly(^2) -- The next to last cubic polynomial in the spline.
```

ComplexVector

The *ComplexVector* class supports these class methods:

from(vector1, vector2)	Creates a complex vector from two equally sized real vectors
zero(size)	Creates a complex vector with *size* ones
random(size)	Creates a complex vector with *size* items, using a uniform distribution
nrandom(size)	Creates a complex vector with *size* items, using a standard normal distribution
new(length, lambda)	Initializes a complex vector using an arbitrary lambda function.

As with real vectors, random complex vectors created by using *random* and *nrandom* are easily scaled and shifted using vector operators.

complexvector::new is similar to **vector**::new, except that the return type of the lambda is expected to be a complex number:

```
complexvector::new(1024, x => x + 2 * x * i)
```

If the lambda function returns an integer or a double, the result is automatically converted to a complex number.

Matrix

These are the class methods for the *Matrix* class:

zero(size)	Creates a square matrix with all cells containing zeros
zero(rows, cols)	Creates a rectangular matrix with all cells containing zeros
eye(size)	Creates a square identity matrix
diag(vector)	Creates a square diagonal matrix, given the diagonal vector.
diag(values...)	Creates a square diagonal matrix, given the values in the diagonal.
random(size)	Creates a random squared matrix, using a uniform distribution.
random(rows, cols)	Creates a random rectangular matrix , using a uniform distribution.
nrandom(size)	Creates a random squared matrix, using a standard normal distribution.
nrandom(rows, cols)	Creates a random rectangular matrix, using a standard normal distribution.
lrandom(size)	Creates a random lower triangular matrix, using a uniform distribution.

lrandom(rows, cols)	Creates a random lower triangular matrix, using a uniform distribution.
lnrandom(size)	Same as above, using a standard normal distribution.
lnrandom(rows, cols)	Same as above, using a standard normal distribution.
rows(vectors...)	Creates a matrix given its rows as vectors.
cols(vectors...)	Creates a matrix given its columns as vectors.
cov(series...)	Calculates the covariance matrix of a list of series.
corr(series...)	Calculates the correlation matrix of a list of series.
new(rows, cols, lambda)	Initializes a rectangular matrix using an arbitrary lambda function.

matrix::new is a very flexible way to create and initialize a matrix, either square or rectangular, using an arbitrary function of its indexes. For instance:

```
matrix::new(5, 5, (r, c) => r * c)
```

The local parameters *r* and *c* in the lambda function are integer parameters.

Model

The *Model* class groups all calls that returns an interactive model. These are the currently supported methods:

compare(series, series)	Compares two series or vectors. It's also available as a global function.
mvo()	Applies the Critical Line Algorithm

2.7 Operators

AUSTRA provides a wide repertoire of operators, in a attempt to match how they are used in everyday algebra and arithmetic.

Arithmetic operators

These operators apply both to integer and real values:

+	Addition. Can be also used as a unary operator.
-	Subtraction and negation.
*	Multiplication.
/	Both real and integer division.
%	Integer remainder.
^	Power: 2^3 = 8, 9 ^0.5 = 3.
'	Complex conjugation, for complex operands.

Most of then may also be used with complex numbers.

Though the power operator works both for integer, real and complex numbers, the compiler optimizes the special cases when the power is 2, 3 and 4, so equalities like this exactly holds:

```
i^2 = -1
```

The multiplication operator can be elided when the first operand is a real or an integer and it is immediately followed by an identifier:

```
2pi = 2 * pi
2x^2 + 3x + 1 = 2*x^2 + 3*x + 1
1/2x = 1 / (2*x)
```

Comparisons

These operators are used for comparing all compatible operands:

=	Equality.
!=	Inequality.
<>	A synonym for the inequality operator.
<	Lesser than.
<=	Lesser than or equal.
>	Greater than.
>=	Greater than or equal.

Comparisons can be fused for numeric operands using the following syntax:

```
sqrt(pi) <= pi <= pi^2
```

Fused ranges only require combining same-direction comparisons. For instance, `<=` and `<` are compatible, but `<` and `>` are not.

Date operators

These operators work for dates:

+	Adds a number of days (integer) to a date (Date).
-	Gets the number of days between two dates.

Series operators

These are the operators allowed with series operands:

+	Adds values from two series. It can also be used to add a numeric value to each value in a series. E.g.: `aapl + msft`, `aapl + 1.0`
-	Subtracts values from two series, as a binary operator. It can also be used to subtract a number from each value in a series. E.g.: `aapl - msft`, `aapl - 1.0`

-	Negates values from a series, as a unary operator. E.g.: `-aapl.rets`
*	Multiplies values from a series by a scalar value. E.g.: `aapl.rets * 1.1`
.*	Pointwise multiplication of the values from two series. E.g.: `aapl.rets * msft.rets`
/	Divides values from a series by a scalar value. E.g.: `(aapl.rets + msft.rets) / 2`

When two series with a different size are combined, values are aligned to the most recent date of both, and the result contains only points common to both series.

Vector operators

Real and complex vectors support these operators:

+	Adds two vectors, or a vector and a scalar. E.g.: `[1, 2, 3] + [3, 4, 5], [1, 2, 3] + 0.5`
-	Subtracts two vectors or a vector and a scalar. E.g.: `[1, 2, 3] - [3, 4, 5], 3.5 - [1, 2, 3]`
-	Negates values from a vector, as a unary operator. E.g.: `-aapl.values`
*	When applied to two vectors, represents the inner vector product. It can also be applied to a vector and a scalar value. E.g.: `[1, 2, 3] * [3, 2, 1] = 10, [1, 2, 3] * 0.5`
.*	Point-wise multiplication of the values from two vectors. E.g.: `[1, 2, 3] .* [3, 2, 1] = [3, 4, 3]`
/	Divides values from a vector by a scalar value. E.g.: `[1, 2, 3] / 2`
'	For complex vector, the complex vector conjugation, item-wise.
^	The outer product of two real vectors, yielding a matrix.

The outer product of vectors x_i and y_i returns the matrix with components $m_{ij}=x_i*y_j$:

```
> [1,2,3]^[4,5,6]
  4   5   6
  8  10  12
 12  15  18
```

Matrix operators

These are the operators available for matrices:

+	Adds two matrices, or a matrix and a scalar.
-	Subtracts two matrices, or a matrix and a scalar.
-	Negates values from a matrix, as a unary operator.
*	Matrix * matrix: matrix multiplication. Matrix * number: matrix scale.

	Matrix * vector: vector transformation. Vector * matrix: vector is transposed and then transformed.
.*	Point-wise multiplication of the values from two matrices.
/	Divides values from a matrix by a scalar value.
\	Solves a system of linear equations. The second operand can be a matrix or a vector.
'	Transpose (suffix operator)

Linear equation solving can be tested this way:

```
let m = matrix::random(5) + 0.01, v = vector::random(5),
    answer = m \ v in
    m * answer - v
```

Internally, the LU factorization of the matrix is used for solving the equation.

Regarding the unary transpose operator, the parser locates expressions like this one in order to optimize them:

```
matrix' * vector
```

This expression is translated directly to a call to `Matrix.TransposeMultiply(Vector)`, which is faster and waste less temporary memory, without losing precision. Of course, the same operation can also be directly written as:

```
vector * matrix
```

The parse also optimizes this:

$$matrix_1 * matrix_2'$$

The above expression is translated to a call to `Matrix.MultiplyTranspose(Matrix)`, which is also faster and needs less memory. This other pattern is also recognized and optimized:

$$matrix * vector_1 + vector_2$$

This expression is translated as `Matrix.MultiplyAdd(Vector, Vector)`, saving some allocations and gaining some time.

2.8 Indexers

Indexers allows access to a single element of a matrix, vector, or series, or to a range of items within them.

Matrices

Individual cells are accessed using the row and column inside brackets:

```
mat[0, 0]
mat[mat.rows - 1, mat.cols - 1]
```

All indexes starts from zero. If the row index is omitted, a whole column is returned:

```
mat[, 0]
```

Omitting the column number yields a whole row:

```
mat[0,]
```

Carets can also be used in any of the two indexes, to count positions from the end. For instance, this expression returns the rightmost lower cell of the matrix:

```
mat[^1, ^1]
```

Columns and rows can also be extracted as vectors using relative indexes:

```
mat[, ^2]  -- Next to last column.
mat[^2,]   -- Next to last row.
```

Ranges are accepted for both dimensions, and can be combined with indexes too:

```
mat[0:^1, 0:^1]  -- Remove last row and last column.
mat[^1, 1:^1]    -- Last row without first and last items.
```

Vectors

Both for real and complex vectors, individual values are accessed using its position, starting from zero, inside brackets:

```
vec[0]
vec[vec.length - 1]
```

A segment can be extracted as another vector by using this notation:

```
vec[1:vec.length - 1]
```

The above expression removes the first and the last element from a vector. The upper bound is excluded.

The caret can be used in indexes and segments, to count positions from the end. For instance, this expression returns the next to last item of a vector:

```
vec[^1]
```

This equality holds:

```
vec[1:^1] = vec[1:vec.length - 1]
vec[^5:^2].length = 3
```

Vectors and series also support safe indexers. With normal indexers, like `v[1000]`, an out-of-range reference throws an exception and interrupts the evaluation of the formula. If braces are used instead of brackets, and out-of-range reference returns `0.0` and it is considered as a valid use. For instance, the following expression returns zero:

```
[1, 2, 3, 4]{1000}
```

Safe indexers are useful when used inside lambda functions.

Series

Points in a series can be access using an index expression between brackets:

```
aapl[0]
aapl[appl.count - 1].value = aapl.last.value
aapl[^2] = aapl[aapl.count - 2]
```

Series also supports extracting a slice using dates or indexes. In the first case, you must provide two dates inside brackets, separated by a colon, and one of the bounds can be omitted:

```
aapl[jan20:jan21]
aapl[jan20:15@jan21]
aapl[jan20:]
aapl[:jan21]
```

The upper bound is excluded from the result. Date arguments in a series index do not support the caret (^) operator for relative indexes.

When using numerical indexes in a slice, the behaviour is similar to the one of vectors:

```
aapl[1:aapl.count - 1].count = aapl[1:^1].count
```

Index

- U -

- V -

- Z -