**Welcome to AUSTRA**

AUSTRA is an efficient mathematical library, written in C# and running on .NET Core, which is also used by a small functional language designed to handle financial series and common econometric models.

Both library and language, also support vectors, matrices, transforms and the most frequently used operations from linear algebra, statistics, and probability.

The library code is hardware-accelerated, using all resources provided by the CPU. The language compiler is also an optimizing compiler, detecting common expression patterns and substituting them by more efficient method calls, whenever possible.

Austra contains three main components:

1. The Austra library, written in C# and .NET Core 8.
2. The Austra language: a simple formula-oriented language for testing and exploring the library.
3. The Austra application: a desktop application, written in WPF for Windows, providing a code editor with syntax highlighting and code completion, for trying the language.

**The design of the library**

The library has been designed as a set of mostly immutable types, for facilitating their concurrent use. Most of the methods are hardware-accelerated, either using managed references, SIMD operations or both. Memory pinning, and raw pointers, have been reduced to the minimum, to ease the garbage collector's work.

Using immutable vectors, series and matrices has one drawback, and it is more stress for the garbage collector. For that reason, we offer combined operations, like other libraries do, to fuse several linear operations into one, when possible. The AUSTRA parser detects most of these cases for optimizing them.

**Vectorization versus tasks**

This will sound unintuitive, but it has been a guide when designing the Austra Library:

Library code should make as much use as possible of hardware vectorization, and only when this way is exhausted, you should turn to task concurrency if it makes sense.

My points:

* Library methods are usually short. For instance, Map is embarrassingly parallel, but even a vector with 2048 items takes around a microsecond to be mapped. That is a very short span to attempt parallelization using tasks: the overhead of starting and waiting for finalization trumps any gains of task parallelism.
* Neither vectorization nor parallelization plays nice with modularity.
* We have chosen, for Austra, applying all possible vectorizations at the lowest level, and leaving task parallelism to higher level abstractions designed by the consumers of the class.

In any case, using task parallelization with Austra is easy, in part due to classes implementing non-mutating operations.

**Linear Algebra**

Austra provides classes for dense vectors and matrices, for double-precision arithmetic. It also features an efficient CVector type. Single-precision floats, complex and sparse matrices are planned for a future sprint. All operations takes advantage of C# operators when possible, so most of the operations are non-destructive.

There are three classes for representing matrices:

* [Matrix](https://marteens.com/austra/library/html/f810f83b-4bd1-527a-337c-8e37ac9416b7.htm) is the general type that you will use most of the time.
* Lower triangular matrices are represented by the [LMatrix](https://marteens.com/austra/library/html/78402da4-8c3d-bcfe-05dd-45f4ba4f0e90.htm) type.
* Upper triangular matrices are represented by the [RMatrix](https://marteens.com/austra/library/html/d275389f-2f69-566b-0cc7-80517181354a.htm) type.

The point with these two additional types is not to save space, since the underlying data structure is the same, but to provide a more efficient implementation of several methods and operators. There are also some logical advantages, regarding type safety since some decompositions returns triangular matrices.

As usual, matrix multiplication has been fully optimized using loop reordering and unrolling, blocking and hardware intrinsics, including fused multiply and add. There are variants for multiplying a matrix by another matrix transposed on-the-fly, for multiplying a vector by a transposed matrix and for accelerating linear combinations of vectors.

All these types are read-only structures, acting as a thin layer above C#'s arrays. Even the storage for a matrix is a one-dimensional array, since multidimensional arrays in .NET are less optimized for bound checking, getting a managed reference and other low-level operations.

**Matrix factorizations**

Austra provides classes for the following matrix factorizations:

* Lower-Upper (LU) Factorization.
* Cholesky Factorization.
* Eigenvalues Decomposition (EVD).

[Solve(DVector)](https://marteens.com/austra/library/html/34cbf708-9295-ddca-555f-8a85223b1870.htm) and [Solve(Matrix)](https://marteens.com/austra/library/html/74812769-36ef-080d-7c70-f94310ec9687.htm) uses the LU factorization internally.

**Time series**

The kernel of Austra was an implementation of the Mean-Variance optimizer. This means that time series were implemented before vectors and matrices.

Series are collections of pairs date/value, and they are sorted by date. Values can be used as vectors, but there are some differences. 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.

**Mean-Variance Optimizer**

A Mean-Variance Optimizer implementation is included ([MvoModel](https://marteens.com/austra/library/html/c7d7b778-6d78-db70-740b-8f9c0d5f143e.htm)). This functionality is available at the formula language via the model::mvo class method.

The MVO model is rendered as an interactive model by the AUSTRA desktop application.

**Polynomials and root finding**

The [Polynomials](https://marteens.com/austra/library/html/f8b6f7e5-d4f9-659b-f3ab-71742d3785ed.htm) static class provides methods for polynomial evaluation and root finding. The [Solver](https://marteens.com/austra/library/html/781755a4-b348-44d0-354d-c95166764292.htm) class implements a simple variant of the Newton-Raphson method for root finding.

There's also a [PolyEval(Complex, DVector)](https://marteens.com/austra/library/html/585d40f4-3977-425e-5559-407b09f9d1aa.htm) for evaluating polynomials using the Horner's method, and a [PolySolve(DVector)](https://marteens.com/austra/library/html/a0ec2f28-67c5-7fcd-1971-11814b1046af.htm) for analytically finding roots whenever possible, and using eigenvalues of the Frobenius matrix in the general case. There's even a [PolyDerivative(Complex, DVector)](https://marteens.com/austra/library/html/98530ce7-93d9-88f1-b7c7-c074bc4e7d11.htm) for computing the derivative of a polynomial at a given abscissa.

Natural cubic splines have also been implemented, both for series and for functions, using a grid. You can even calculate the derivative of a spline at any point in the supported range.

**Fast Fourier Transform**

Austra implements a decent FFT algorithm, compared to most popular managed implementations. It uses the Cooley-Tukey algorithm, and it's optimized for small sizes. Small primes are handled either with Bluestein's or Rader's algorithm, depending on the size.

In any case, there is still room for improvement, and it's planned to be optimized in the future. AVX prefers structs of arrays over arrays of structures, and this preference obviously applies to complex arithmetic: it's more efficient to represent the real and the imaginary parts of a list of complex numbers in separate arrays.

**Language goals and design**

One of the motivations for creating Austra was having an easy-to-use language for testing and exploring functionality.

* The language should be mostly a functional one. Functional languages are expression-oriented, concise and discourages mutability. These features match very well the characteristics of the library.
* On the other hand, we did not want a complicated language with lazy evaluation and monads. I really like monads! Some of my best friends are monads! Jokes aside: I want Austra to be used by a wide base of professionals, instead of a selected group of freaks.
* A problem with R and Matlab, which loosely fall in the same category as Austra, is the pollution of the global namespace. We wanted to avoid that. Instead of having a global product function that you could apply to a vector, we prefer a product method that is a feature of vectors.

There is also an important non-goal:

* We are not trying to substitute C# with Austra. AUSTRA, the language, is not supposed to be a Turing-complete programming language.

These are some consequences from the non-goal:

* We do not intend writing the Austra library in AUSTRA. That may be the goal for a next step, and, indeed, we have already some ideas and plans to do it. It would require, for make any sense, automatic vectorization, for example.
* The type system of AUSTRA is very simple. There are no generic types. Type inference is primitive. Only a handful of classes from the library are fully exposed. And, in the current version, we still have no support for tuples.

To diminish the complexity of using the language, we also conceal some types as much as possible, to reduce the number of class names the programmer has to remember. The language defines a small set of classes on which class methods, i.e., constructors and static methods, can be called. These classes are:

* math, for grouping global functions and variables.
* matrix, for dealing with all kinds of matrices.
* vec, cvec and nvec, for real vectors, complex vectors, and integer vectors.
* seq, cseq and nseq, for real sequences, complex sequences, and integer sequences.
* series, for time series.
* spline, for cubic splines.
* model, for mathematical models and tools.

Of course, AUSTRA handles a long list of types, from primitive types such as date and int, to classes generated by transforms or matrix factorizations.

# Language overview

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 probabilities.

AUSTRA formulas 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 topic introduces the basic syntax of the language.

## Lexical syntax

The lexical syntax of AUSTRA is very similar to most programming languages:

* White space, including line returns, are completely ignored.
* Identifiers and keywords are key insensitive.
* Unicode characters are allowed in identifiers. So, yes: τ = 2\*π is a valid expression. Of course, pi is also allowed, and the code editor helps while typing Greek characters.
* Semicolons (;) are mandatory for separating statements, but not as statement terminators.

#### Numeric literals

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

**Austra**Copy

2023;

1.0;

-0.1E-16

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

**Austra**Copy

2.0i;

-3i

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

**Austra**Copy

1-3i = 1 - 3 \* i

Complex numbers can also be created using the complex function:

**Austra**Copy

complex(1, -3) = 1 - 3 \* i;

complex(3) = 3 + 0i

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

Complex can also be built using the polar notation:

**Austra**Copy

polar(1, pi/2) -- Another way to write the imaginary unit.

#### String literals

String literals are enclosed by double quotes and cannot cross line boundaries.

**Austra**Copy

"A simple string literal";

"A string literal with a quote: ""Wow!"". That was the quote."

#### Date literals

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

**Austra**Copy

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:

**Austra**Copy

jan20; -- January 1st, 2020

may42 -- May 1st, 1942

A day can be added using this syntax:

**Austra**Copy

6@jan20;

31@jul2021

#### 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:

**Austra**Copy

-- A verbose version of math::min()  
**if** aapl.mean < msft.mean **then** aapl.mean -- Another comment.  
**else** msft.mean

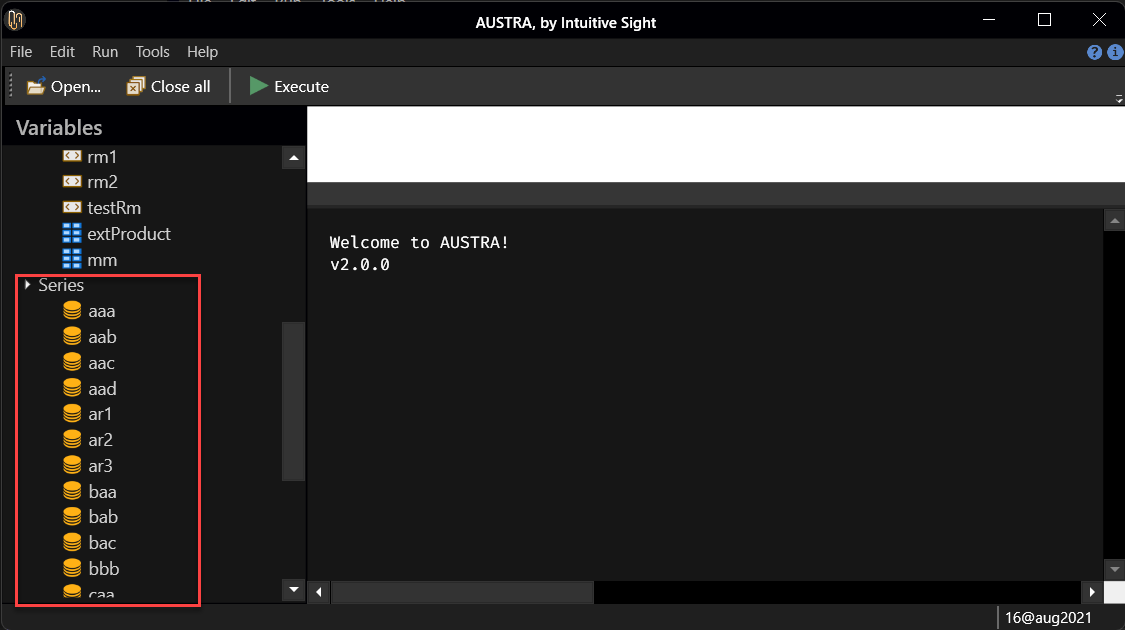
## Root objects

Every AUSTRA expression must start with a root object. It could be either a global variable, a local variable, a class method, a class variable, or a code definition.

#### Global variables

Global variables come in two flavours: persistent variables and session variables. Persistent variables come mostly from an external source, like a JSON file, a database, or an external service. In this AUSTRA version, those persisted variables are always time series, because they have a predictable serialization format. This design decision, of course, may change at some point of the evolution of the library.

For instance, when I open the AUSTRA application in my system, it automatically loads a set of series and definitions that are stored in a subfolder Austra of my Documents folder, in a file named data.austra, and its main windows looks something like this:

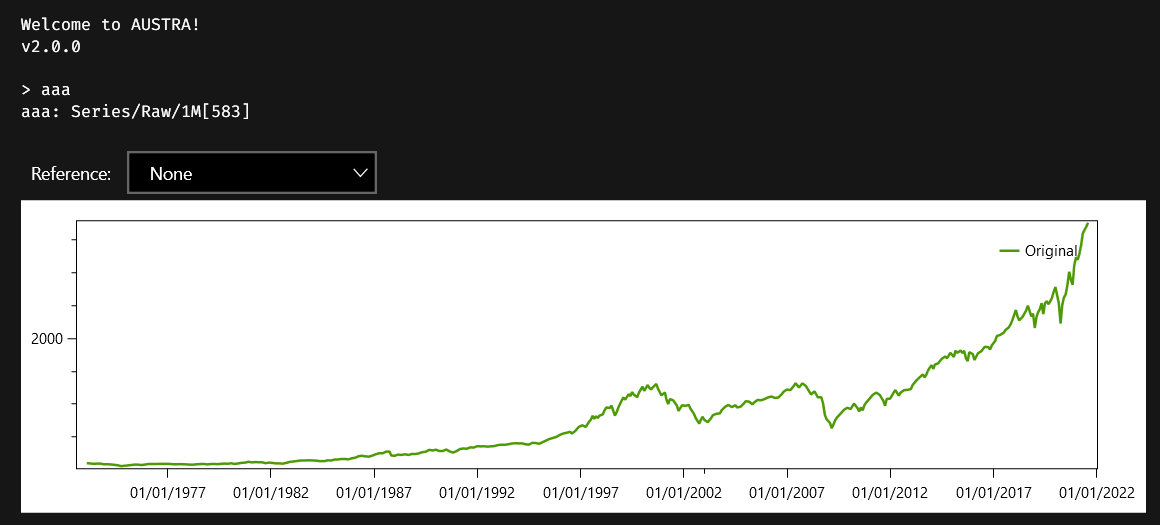


Persistent series are shown below a Series node. I can type the name of any of these variables in the Code Editor:

**Austra**Copy

aaa

When I press F5, AUSTRA translates the expression and immediately shows the content of the aaa series:



#### Session variables

Session variables, as the name indicates, are defined inside a user session, and die with the session. They are defined and removed using the set statement:

**Austra**Copy

**set** v1 = [1, 2, 3, 4, 5];

**set** v2 = v1.map(x => 1 / x);

v2.plot;

-- v1 is removed now:  
**set** v1;

-- v2, however, persist for the rest of the session.

v2.plot

Only the value of the variable is stored, but not the formula that was used to calculate that value. This means, for example, that every use of the session variable will return the same value, even if the value was created using a random number generator:

**Austra**Copy

-- v1 is created using random numbers:  
**set** v1 = **vec::**random(10);

-- Every use of v1 returns always the same vector:

v1 = v1;

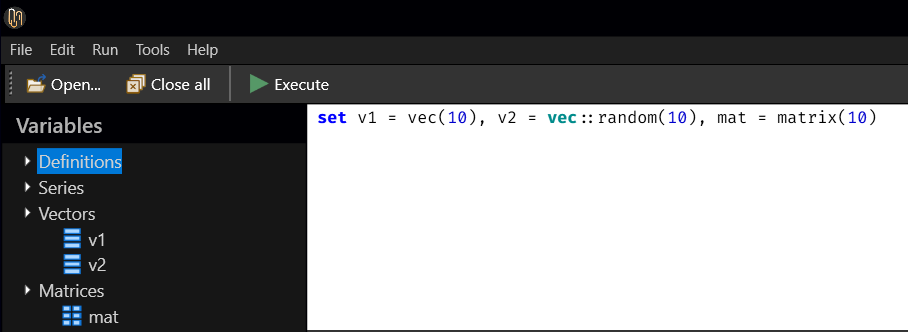
-- This is in contrast with the behavior of local variables.  
**let** v2 = **vec::**random(10);

-- This expression will return **false**:

v2 = v2

Local variables are explained [here](https://marteens.com/austra/library/html/8b381718-bf08-4762-a51b-1516af27bef2.htm).

Session variables appears in the **Variables** panel, each one inside a node according to their types:



#### Class methods and class constants

Class methods in AUSTRA correspond both to constructors and static methods in traditional OOP languages, like C#.

Let's start with some variables:

**Austra**Copy

i = **math::**i;

e = **math::**e;

pi = **math::**pi **and** pi = **math::**π

The same equivalence is valid for what we normally would consider "global functions":

**Austra**Copy

exp(π\*i);

**math::**exp(**math::**pi \* **math::**i)

Those global functions and constants are considered as belonging to the math for avoiding problems if any of these symbols is redefined as a persistent or session variable.

Of course, there are more classes than math, and we can use their class methods for creating new objects:

**Austra**Copy

**matrix::**random(10);

**vec::**new(10);

vec(10)

As the last example shows, when you call a new method on a class, you can omit the ::new part and use just the class name as synonym.

# Definitions

Code definitions are formulas saved for future use. They are saved and loaded from any persistent storage used by AUSTRA. You can define either parameter-less definitions, that act like macros, or parametric definitions, which are the equivalent of user-defined functions.

## Creating definitions

Definitions are created using the def statement:

**Austra**Copy

**def** cxMvo = **model::**mvo(sm\_ret, sm\_cov, sm\_low, sm\_high)

A description can be associated to a definition using the following syntax:

**Austra**Copy

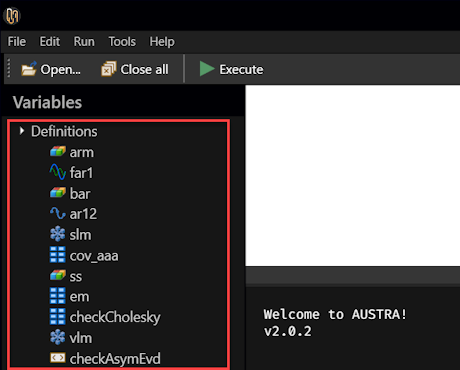
**def** cxMvo:"MVO Model" = **model::**mvo(sm\_ret, sm\_cov, sm\_low, sm\_high)

Removing an existing definition is achieved with the undef command:

**Austra**Copy

**undef** cxMvo

In the AUSTRA desktop application, definitions appear in the **Variables** panel, inside a **Definitions** node:



#### Definitions cannot use session variables

Code definitions must respect some limitations. The most important one is that they cannot reference session variables. This sequence of commands is invalid:

**Austra**Copy

**set** vector = [1, 2, 3, 4];

**def** fact4 = vector.product -- Invalid code definition.

The reason behind this constraint is that session variables only store their current values, but not the formula that generated that value.

#### Definitions may use existing definitions

A code definition may refer to an existing definition. For instance:

**Austra**Copy

**def** sm\_cov = **matrix::**covariance(aapl, msft, hog, dax);

**def** sm\_ret = [1, 0.9, 1.2, 0.8];

**def** cxMvo = **model::**mvo(sm\_ret, sm\_cov, vec(4), **vec::**ones(4))

In this case, removing either sm\_cov or sm\_ret, would also remove cxMvo.

#### Deterministic callings

Let's say we make this definition:

**Austra**Copy

**def** extProduct = **vec::**random(4) ^ **vec::**random(4)

This definition calls twice a class method that creates a random vector. The caret operator, ^, combines those two vectors in a 4x4 matrix. Executing these definitions two times in a row gives, as expected, different results:

**Austra**Copy

> extProduct

ans ∊ ℝ(4⨯4)

0.416065 0.493621 0.412334 0.0249965

0.390261 0.463007 0.386762 0.0234462

0.377909 0.448353 0.37452 0.0227041

0.49103 0.58256 0.486626 0.0295002

> extProduct

ans ∊ ℝ(4⨯4)

0.0251534 0.0182728 0.0452763 0.00933612

0.0374942 0.0272379 0.06749 0.0139167

0.0555746 0.0403725 0.100035 0.0206275

0.0256057 0.0186015 0.0460906 0.00950403

That is the expected behaviour. However, this could be inconvenient to test properties of the result. For instance, we could want to check the determinant of the product, or that a double transpose works fine:

**Austra**Copy

extProduct = expProduct''; -- Double transpose.

(extProduct \* extProduct).det - extProduct.det^2

AUSTRA assumes that, inside a formula, all parameter-less definitions call must return the same value. For that purpose, the two above formulas are internally rewritten as:

**Austra**Copy

**let** x = extProduct **in** x = x'';

**let** x = extProduct **in** (x \* x).det - x.det ^ 2

A local variable is created under the hood for evaluating the definition just once inside the current formula.

This automatic caching only takes place for parameter-less definitions. If you want to disable this behaviour, just add an exclamation sign right after the definition identifier, when using the definition:

**Austra**Copy

-- This first expression returns **true**.

extProduct = extProduct;

-- This second expression returns **false**.

extProduct = extProduct!;

-- This expression also returns **false**.

extProduct! = extProduct

#### Function definitions

A definition can also have parameters, for defining a function. For instance, the factorial of an integer can be defined this way:

**Austra**Copy

**def** fact(n: **int**) = iff(n <= 1, 1, [2..n].prod)

The above definition is non recursive. Recursive functions must declare their return type:

**Austra**Copy

**def** recFact(n: **int**): **int** =

**if** n <= 1 **then** 1 **else** n \* recFact(n - 1)

You can use local variables when defining a function:

**Austra**Copy

**def** mcd(a, b: **int**): **int** =

**let** m = a % b **in** iff(m = 0, b, mcd(b, m))

And you can also define auxiliary functions inside a function definition:

**Austra**Copy

**def** fact(n: **int**) =

**let** f(n, acc: **int**): **int** = iff(n <= 1, acc, f(n - 1, n \* acc)) **in**

f(n, 1)

In this case, the inner function f is the one that is directly recursive. The outer function does not need to declare its return type.

**Local variables**

AUSTRA is a functional language, so it has a functional technique for declaring what in a procedural language would be temporal or local variables.

**LET clauses**

The functional technique for declaring local variables in a formula is the **let** clause.

**Austra**Copy

**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 being evaluated. As the example shows, each variable defined in the let clause can use any of the previously declared variables in the same clause.

**Script-scoped LET clauses**

When writing several statements in a script, let/in clauses are valid only for the statement they precede, but not for other statements:

**Austra**Copy

**let** m = **matrix::**lrandom(5),

m1 = m \* m',

c = m1.chol **in**

(c \* c' - m1).aMax;

-- The next statement cannot use "m".

m

If you need a local variable to be available for all statements that follow in a script, you must use a variant of let which does not terminate with an in keyword, but with a semicolon:

**Austra**Copy

**let** m = **matrix::**lrandom(5);

-- Now, "m" is available for the rest of the script.  
**let** m1 = m \* m',

c = m1.chol **in**

(c \* c' - m1).aMax;

-- The next statement is valid.

m

**Note**

Some functional languages, as Haskell, feature another construct for abstracting sub-expressions. Haskell, for instance, offers both **let** and **where**. **let** is located before the expressions that make use of it, and **where** comes after the main expression.

In AUSTRA, we prefer let, for the sake of Code Completion. So far, I cannot think of any use for **where** that cannot be solved better with let.

**Local function definitions**

Functions can be defined in let clauses. For instance:

**Austra**Copy

**let** mcd(a, b: **int**): **int** = **if** a % b = 0 **then** b **else** mcd(b, a % b) **in**

mcd(80, 140)

In the above example, the function is defined in a let/in clause, but it could also be defined as a script-scoped local function.

**Note**

Since mcd is recursive, its return type must be declared in the function header.

Function definitions may have their own local variables, as in this variant of the above example:

**Austra**Copy

**let** mcd(a, b: **int**): **int** =

**let** m = a % b **in** iff(m = 0, b, mcd(b, m)) **in**

mcd(80, 140)

This way, we save one evaluation of the remainder.

Local functions may also be declared inside other functions. For instance, this code defines a function for the factorial, but uses an intermediate function that can be evaluated using tail recursion, for efficiency:

**Austra**Copy

**let** fact(n: **int**) =

**let** f(n, acc: **int**): **int** = iff(n <= 1, acc, f(n - 1, n \* acc)) **in**

f(n, 1);

fact(10);

Please note that the in keyword applies to the right-side of the definition of factorial. The let clause that defines factorial, on the contrary, is a script-level clause, with no associated in.

# Lambda functions

Lambda functions are inline-defined anonymous functions that can be used as parameters in normal methods and class method calls.

## Lambda functions with one parameter

Series and vectors, for instance, has an all method to check if all their numeric values satisfy an arbitrary condition. The condition is the only parameter of the method and must be passed as a lambda function. Let's say we have an aapl\_prices persistent variable holding a series of prices. We can verify that all those prices are positive using this formula:

**Austra**Copy

aapl\_prices.all(x => x >= 0) -- It should return **true**.

The above formula checks whether all values in the price 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:

**Austra**Copy

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:

**Austra**Copy

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

Notice that in this new example, we have used another name for the "arbitrary given value": value instead of x. This renaming has no effect in the formula.

This example shows how to use the related method any:

**Austra**Copy

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:

**Austra**Copy

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

In all cases, the type of the parameter of the lambda is determined by the method the lambda is passed, and so is the returned type. AUSTRA adds any required conversion, as when a double is required for the result and an integer expression is being returned. Regarding the name of the lambda's parameter, you can use any name you like, keeping in mind that it will shadow any predefined identifier inside the lambda function's body.

#### Function names as lambdas

In many cases, you need a lambda that takes a single parameter to transform it into another value from the same type. For instance, the sine function can be approximated using a spline over a uniform grid like this:

**Austra**Copy

**let** s = spline(0, 2\*pi, 1024, x => sin(x)) **in**

s[pi/4]

The above code can be shortened to this:

**Austra**Copy

**let** s = spline(0, 2\*pi, 1024, sin) **in**

s[pi/4];

Or even this if you need to qualify the function name for any reason:

**Austra**Copy

**let** s = spline(0, 2\*pi, 1024, **math::**sin) **in**

s[pi/4];

Since sin is a mono-parametric function and no parameters are supplied, the compiler understands that the function must be used to create a mono-parametric lambda, returning a real value.

## Lambda functions with two parameters

Some methods require lambda arguments with more than one parameter. When a lambda requires two or more parameters, their names must be enclosed inside parenthesis, and must be separated by commas.

That is the case of the zip method, from series, vectors, and sequences, that combines two data samples into one:

**Austra**Copy

aapl\_prices.zip((x, y) => max(x, y))

zip can act on arguments with different lengths, so it only acts in the common part of both. It generates a new series, vector or sequence, and each item will be the combined value created by the lambda function. In the above example, it will be the maximum price for each common date.

## Captured variables

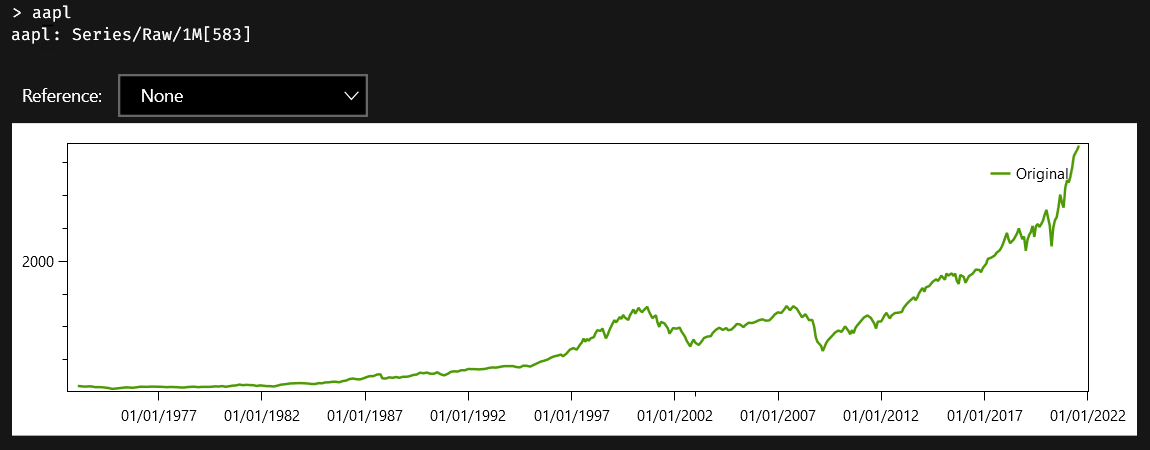
The ncdf() method of a [series](http://localhost:12345/html/1800e13a-baf2-48f8-aefa-746082fe23df.htm) takes a real value and classifies it according to its position in the normal distribution implicitly defined by the series. By definition, it is a value between 0 and 1. Even better, ncdf() is monotonic: 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:

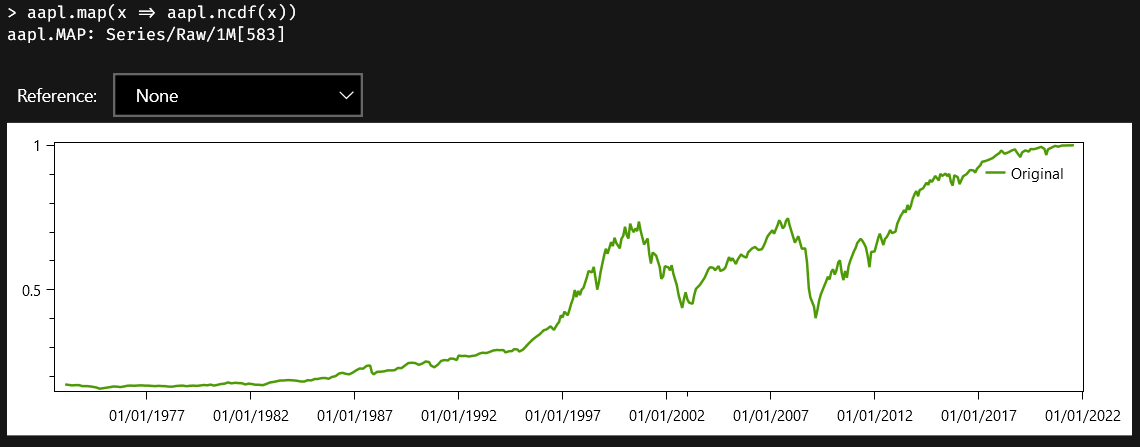
**Austra**Copy

aapl.map(x => aapl.ncdf(x))

Nothing remarkable here: aapl 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 series:



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:

**Austra**Copy

aapl.rets.map(x => aapl.rets.ncdf(x))

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

**Austra**Copy

**let** a = aapl.rets **in**

a.map(x => a.ncdf(x))

Though a is a local variable defined in the main body of the formula, we still can reference it from our nested lambda function. This way, we avoid recalculating the returns of the series in the lambda's body.

**Note**

The series.ncdf(x) method assumes that values in the series can be described by a normal distribution. This is almost never true.

A most useful related method is series.movingNcdf(points), which calculates the ncdf for each value in the series, but calculates the two parameters that defines a normal distribution from a configurable interval of points preceding each calculation.

## Nested lambdas

Another kind of capture takes place when a lambda function is defined inside another lambda. This formula finds all prime numbers up to 100, and uses nested lambdas:

**Austra**Copy

iseq(2, 100).filter(**x** => iseq(2, **x** - 1).all(**div** => **x** % **div** != 0))

**Note**

The above code also uses sequences for generating a range or list of integers.

The underlined text is a definition of a lambda that is being used as the argument of the filter method. It's a function with a single parameter x. Note, however, that inside that lambda, we call another method that has its own lambda function, using the parameter div. The inner lambda can use both its own parameter div, but it also can use x, defined by the outer function.