
THE AUSTRA LENGUAGE
Ian Marteens

Contents

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 might 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 complex vector 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` is the general type that you will use most of the time.
- Lower triangular matrices are represented by the `LMatrix` type.
- Upper triangular matrices are represented by the `RMatrix` 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).

Linear equation solving 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, starting with the `MvoModel` class. 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` static class provides methods for polynomial evaluation and root finding. The `Solver` class implements a simple variant of the Newton-Raphson method for root finding.

There's also a `PolyEval` for evaluating polynomials using the Horner's method, and a `PolySolve` for analytically finding roots whenever possible, and using eigenvalues of the Frobenius matrix in the general case. There's even a `PolyDerivative` 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 must 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`, `bool` and `int`, to classes generated by transforms or matrix factorizations.

Arrays in AUSTRA

One example about how AUSTRA tries to hide complexity from the user is how arrays are handled by the language. Arrays pervades the library. You need an array of reals to create a vector, an array of vectors to create a matrix, and another array of series to create a covariance matrix. But arrays do not match well with a functional programming style.

What AUSTRA does is accepting a variable number of parameters wherever a method needs an array parameter. This is, for instance, how AUSTRA creates a covariance matrix from a list of series:

```
matrix::cov(aaa, aab, aac);
matrix::cov(aaa, aab, aac, aad);
```

And this is how we efficiently create a linear combination of three series, including an "intercept", that is, a constant additional term in the linear combination:

```
series::new([0.1, 0.2, 0.3, 0.4], aaa, aab, aac);
```

In both cases, the implementing code receives an array of series as its last parameter, and AUSTRA automatically gathers all series at the end of the method call in a single array.

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:

```
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` function:

```
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:

```
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.

```
"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:

```
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 to the constant core using this syntax:

```
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:

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

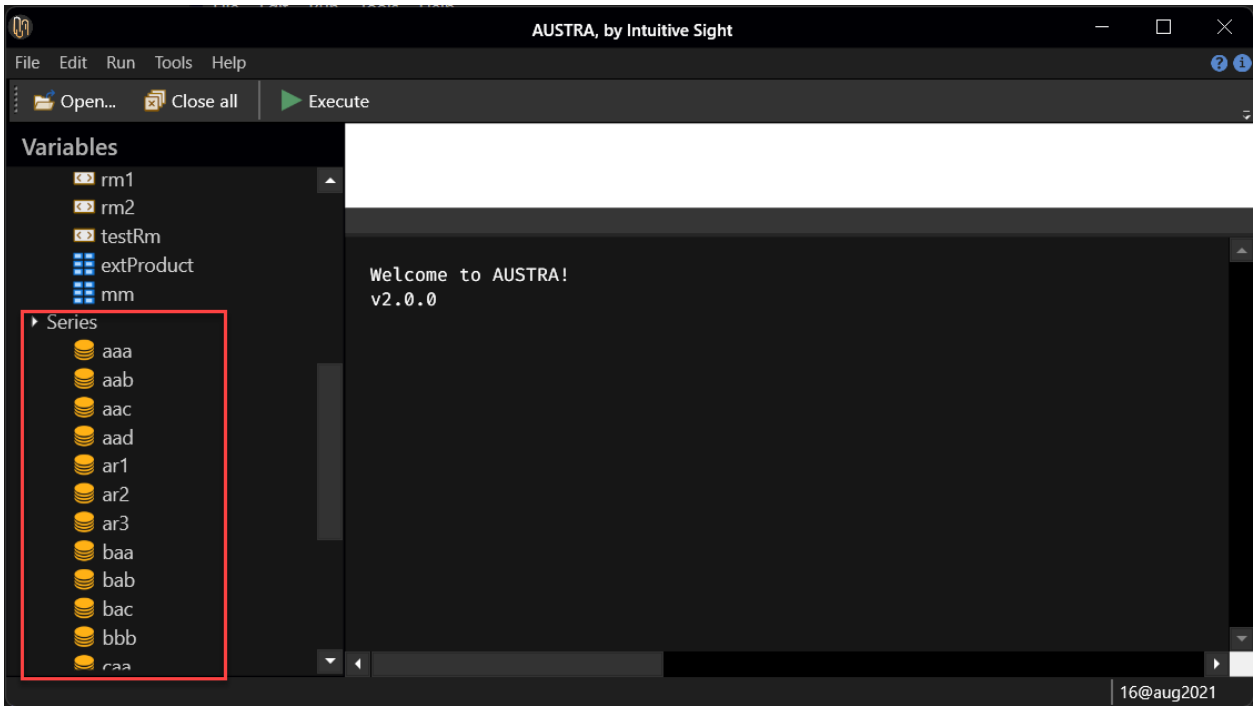
Root objects

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.

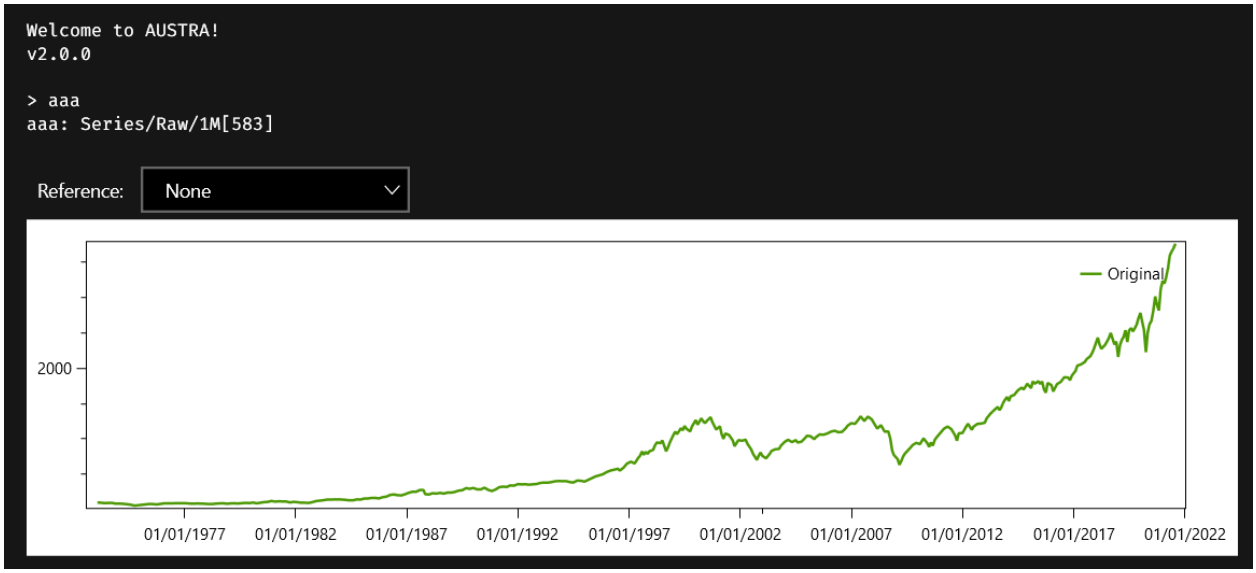
For instance, when I open the AUSTRA application in my system, it automatically loads a set of series and definitions that are stored in a subfolder *Austra* of my *Documents* folder, in a file named *data.austra*, and its main windows looks something like this:



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

```
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:

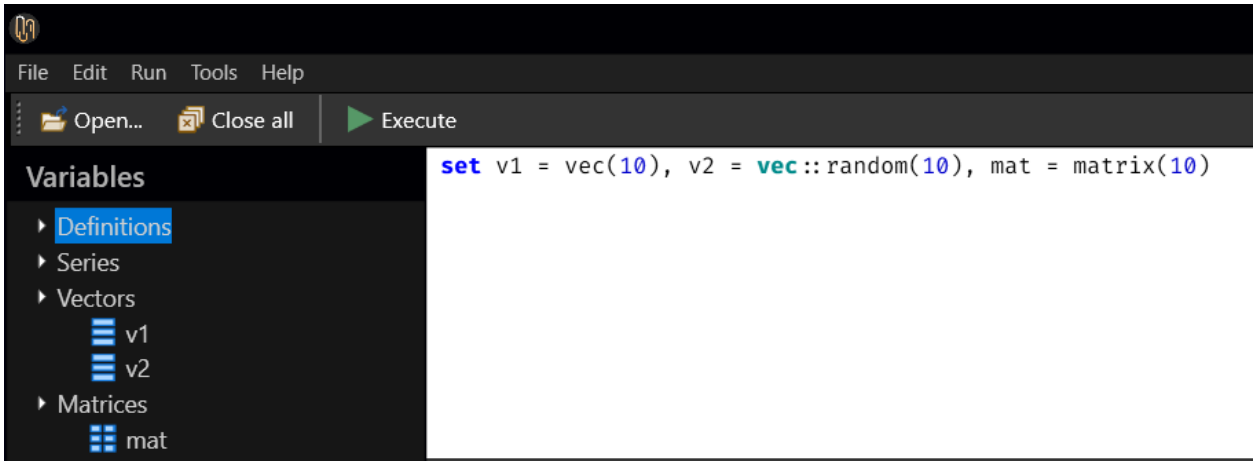
```
set v1 = [1, 2, 3, 4, 5];
set v2 = v1.map(x => 1 / x);
v2.plot;
-- v1 is removed now:
set v1;
-- v2, however, persist for the rest of the session.
v2.plot
```

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:

```
-- v1 is created using random numbers:
set v1 = vec::random(10);
-- Every use of v1 always returns 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 in the next section.

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:

```
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":

```
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:

```
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.

Primitive types

AUSTRA ARITHMETIC IS basically the same as on most programming languages. The language supports:

- 32 bits integers, represented by the `int` type.
- 64 bits integers, represented by the `long` type.
- 64 bits double-precision reals, represented by the `double` type.
- 2x64 bits double-precision complex values, represented by the `complex` type.

Smaller arithmetic types are automatically converted to bigger types when required: `int` to `double`, `double` to `complex`, and even `int` to `complex`. Double values can be converted into integer values using the `toInt` property, as in `pi.toInt`.

Operators

These are the operators available for integers and reals:

`+`	Addition. Can also be used as a unary operator.
`-`	Subtraction. Can also be used as a unary operator for negation.
`*`	Multiplication.
`/`	Both real and integer division.
`%`	Both integer and real remainders.
`^`	Power: `2^3 = 8`, `9^0.5 = 3`.

Most of them 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)
```

AUSTRA also recognizes a superscript *2* (`²`) as an operator to square a value:

```
2x² + 3x + 1 = 2*x^2 + 3*x + 1
```

The AUSTRA code editor simplifies typing this operator with the key's combination (CTRL+G, 2).

Comparisons

These operators are used for comparing all compatible operands:

`=`	Equality.
`!=`	Inequality.
`<>`	A synonym for the inequality operator.
`<`	Less than.
`<=`	Less than or equal to.
`>`	Greater than.
`>=`	Greater than or equal to.
`<-`	Belongs to. Right side must be a vector or a sequence.
`∈`	A fancy synonym for the membership operator (`<-`).

The membership operator can be used with sequences, vectors, matrices, and series:

```
34 <- [1..100];
0 <- vec::random(1024)
```

When the right side of the membership operator is a time series, the left operand may be either a real or a date:

```
0.0 <- appl.rets;
1@jan2020 <- appl
```

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

```
sqrt(pi) <= pi <= pi²
```

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

Complex properties and operators

When you have a complex value in your hands, you can drill into it using a dot and a property name, to extract information about the poor little value:

`real`	The real part of the complex.
`imaginary`	The imaginary part of the complex.
`magnitude`	A magnitude, i.e., the distance to `complex(0, 0)`.
`phase`	The phase, in radians.

```
let c = complex(3, 4) in
    c = c.real + c.imaginary * i
```

If typing `magnitude` is too hard for your nerves, you can use `mag` as an accepted synonym. `real` can be shortened to `re`, and `imag` and even `im` can be used instead of `imaginary`. Since my heart is cold and empty for `phase`, on the other hand, there is no diminutive for that fellow.

In addition to the usual operators, there's a suffix operator for conjugating a complex value:

`'`	Unary suffix operator for complex conjugation.

The `'` operator is also used for transposing a matrix.

Integer properties

Integer values support the `even` and `odd` properties for easy testing of parity:

```
iff(e.toInt.even, "Truncated to 2", "Rounded to 3")
```

The math class

The `math` class groups methods and properties dealing with arithmetic operations. Most of these features come straight from the C#'s `Math` and `Complex`, but it also incorporates other functions that are used in statistics and probabilities.

Our `math` is special in that the class prefix is assumed when not present in a function or property call:

```
-- Write like this, if you are a sucker for pain.
math::sin(math::pi/4) = math::sqrt(2)/2;
-- Standard people use this style.
sin(pi/4) = sqrt(2)/2
```

Why, for the love of Mike, have we sunken all those definition inside the `math` class? It is easy to explain with two points: we did not want to pollute the global name space with lots and lots of symbols, in the first place. Some mathematically oriented languages do just this: everything is a global function, so, at some point, you must come with very clever but cryptic names for your own stuff. Nonetheless, we can omit the class name for the most used names. The second point is related: somebody can shadow one of these global names, such `i` or `max`. In those cases, you still have the long and winding road of prefixing the shadowed name with its class name, and nothing is lost.

These are the methods or functions provided by this class. Most of them work both with integer, real and complex parameters:

`abs`	Absolute value
`acos`	The angle whose cosine is the specified parameter.
`asin`	The angle whose sine is the specified parameter.
`atan(x)`, `atan(x, y)`	The angle whose tangent is the specified parameter. The version with two parameters is equivalent to Atan2.
`beta(x, y)`	Bi-parametric Euler integral of the first kind.
`cbrt`	Cubic root.
`complex`	Creates a complex number from one or two real values.
`cos`	The cosine function.
`cosh`	The hyperbolic cosine function.
`erf`	The error function.
`exp`	The exponential function.
`gamma`	The gamma function: an extension of factorials for real numbers.
`lnGamma`	The natural logarithm of the gamma function.
`log`	The natural logarithm function.
`log10`	Base 10 logarithms.
`max`	The maximum of its two parameters. It also works with dates.

`min`	The minimum of its two parameters. It also works with dates.
`ncdf`	Normal cumulative distribution function.
`polar`	Creates a complex from its circular coordinates.
`probit`	The inverse of the cumulative of the standard normal distribution.
`round(d)`	Rounds a real to the nearest integer.
`round(d, i)`	Rounds a real to the given number of decimals.
`sign`	Returns the sign of the argument.
`sin`	The sine function.
`sinh`	The hyperbolic sine function.
`sqrt`	The square root.
`tan`	The tangent function.
`tanh`	The hyperbolic tangent function.
`trunc`	Truncates a real value.

These are the properties (parameter-less functions) and constants provided by `math`:

`e`	Euler's constant.
`i`	The imaginary unit.
`nrandom`	A random number from the standard normal distribution.
`pearl`	An Easter Egg. Just try me!
`pi, π`	Don't be irrational: be transcendent.
`random`	A random number from a uniform distribution between 0 and 1.
`tau, τ`	Twice π.
`today`	The current date.

Polynomials and solvers

These methods are also defined inside the `math` class, so they can be used without writing explicitly the class prefix:

`solve`	A simple Newton-Raphson solver. See below for details.
`polyEval`	Evaluates a polynomial given a real or complex argument.
`polyDerivative`	Evaluates the first derivative of a polynomial at a real or complex argument.
`polySolve`	Calculates all the roots of a polynomial.

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 `solve` 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`.

The `polyEval` function takes either a complex or a real as its first argument, 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:

```
let x1 = complex(-1, sqrt(2)), x2 = x1';
-- 1, 2, 3 represents the polynomial x² + 2x + 3
polyEval(x1, 1, 2, 3);
-- Coefficients can be grouped in a vector.
polyEval(x2, [1, 2, 3]);
```

The inverse of `polyEval` is the `polySolve` function. It 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 reals. For instance:

```
polySolve(1, 2, 3)
```

You can check the accuracy of the answers from the solver using this trick:

```
let poly = [1, 2, 3] in
  polySolve(poly).all(c => abs(polyEval(c, poly)) <= 1e-15)
```

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:

```
let 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.

Dates

Dates in Austra are represented by the `date` type and stores the number of days since Jan 1st, 1900. Dates support these properties:

`day`	Gets the day of month, starting by 1.
`dow`	Gets the day of the week.
`isLeap`	Is the year from the date a leap one?
`month`	Gets the month of the date, starting with 1.
`year`	Gets the year of the date.

These two methods allow adding either a positive or a negative number of months or years to a date:

`addMonths`	Adds a positive or negative number of months to a date.
`addYears`	Adds a positive or negative number of years to a date.

Adding or subtracting days from a date is achieved with these operators:

`+`	Adds days to a date. The left operand must be a date.
`-`	Subtracts days from a date. The left operand must be a date. It can also be used to find the difference in days between two dates.

Logical values

Logical values are represented by the `bool` data type. Variables and parameters from this type hold one of these two constants: either `false` or `true`.

Operators acting on logical values resembles more the good-old Pascal operators than the C/C++/C# one. It's a matter of personal preference, of course, but also of readability:

`not`	Logical negation.
`and`	Logical conjunction.
`or`	Logical disjunction.

Conditional expressions

Since AUSTRA is a functional language, it doesn't have "statements". However, it provides an `if/then/else` ternary operator, equivalent to the also included `iff()` function:

```
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 written using `iff()` this way:

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

Most of the times, the more verbose ternary operator is easier to read. The ternary operator has another advantage: you can chain several conditions and responses using the `elif` keyword.

```
let x = random;
if x < 0.1 then "Too low!"
elif x < 0.5 then "A little low"
elif x < 0.9 then "A little high"
else "Too high!"
```

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:

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

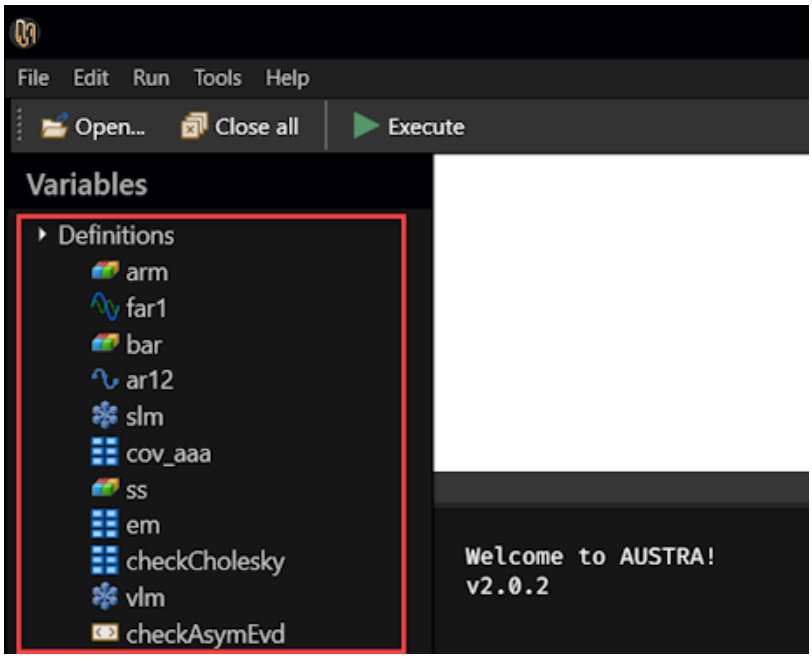
A description can be associated to a definition using the following syntax:

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

Removing an existing definition is achieved with the `undef` command:

```
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:

```
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:

```
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 calls

Let's say we write this definition:

```
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:

```
> 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:

```
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:

```
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:

```
-- 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:

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

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

```
def recFact(n: int): int =
    if n <= 1 then 1 else n * recFact(n - 1)
```

You can use local variables when defining a function:

```
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:

```
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.

```
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:

```
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:

```
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:

```
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:

```
let mcd(a, b: int): int =
    let m = a % b in iff(m = 0, b, mcd(b, m)) in
        mcd(80, 140)
```

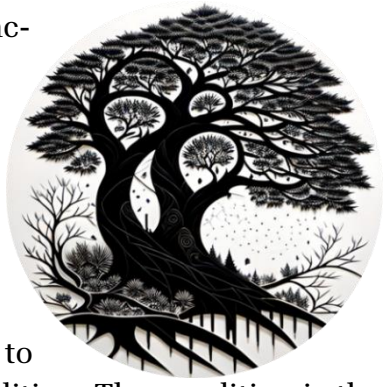
This way, we save one evaluation of the remainder.

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:

```
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:

```
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:

```
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 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`:

```
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))
```

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:

```
let s = spline(0, 2*pi, 1024, x => sin(x)) in
    s[pi/4]
```

The above code can be shortened to this:

```
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:

```
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:

```
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.

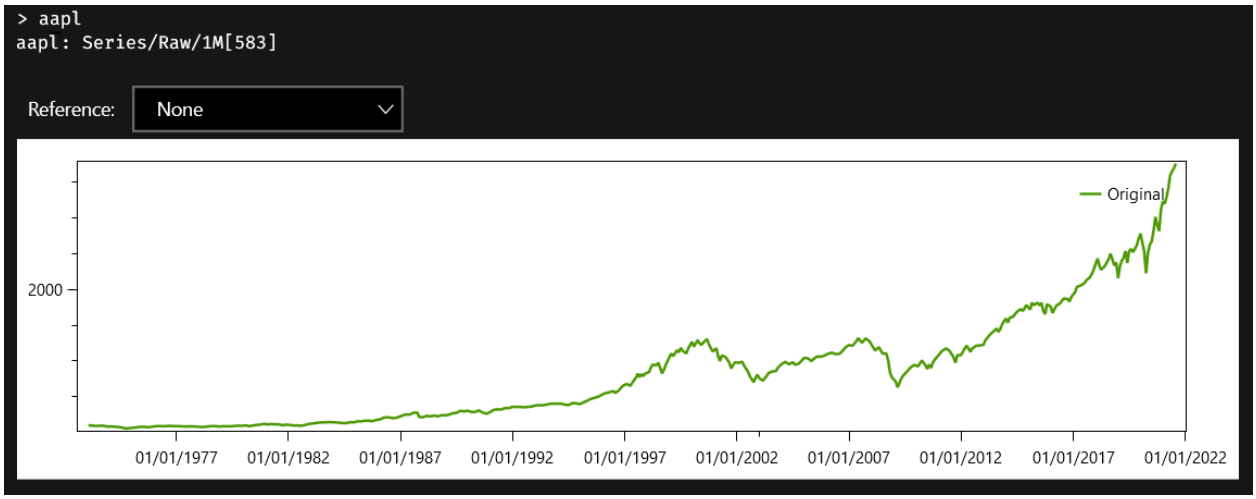
Captured variables

The `ncdf()` method of a series 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.

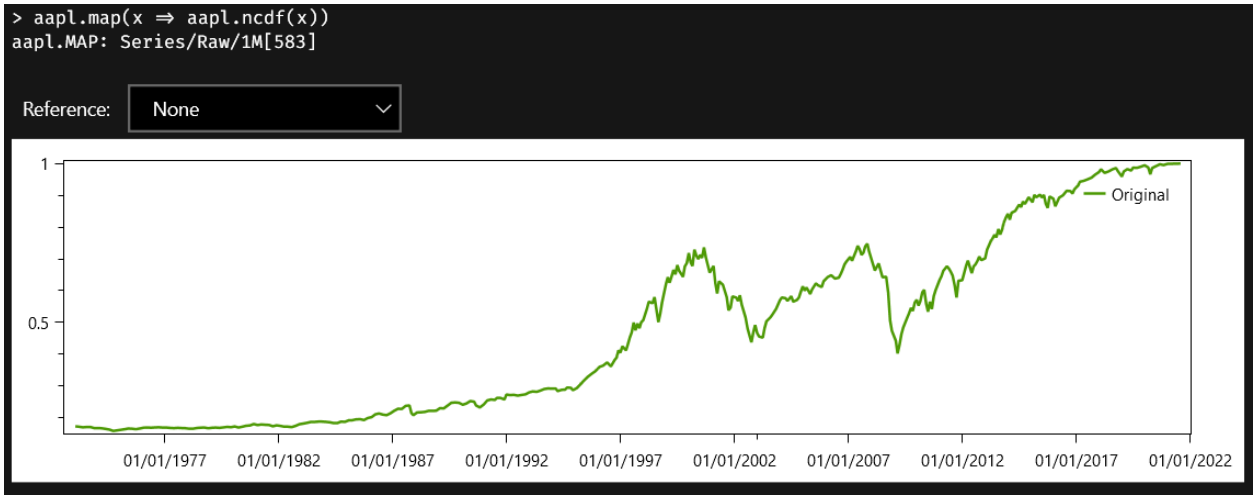
This formula does the trick:

```
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:

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

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

```
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:

```
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.

Time series

THE MOST IMPORTANT data type in AUSTRA is the *time series*: a sorted collection of pairs date/value.

Series come from external sources

Since time series represents data from the real world, most of the times, time series come from persistent variables, that can be stored in an external file or database, and may be periodically updated, either by AUSTRA or by another process.

Additional information in series

Since one of the goals of AUSTRA is to deal with financial time series, there is a few 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. A ticker 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.

Series 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.

Class methods

There is only one constructor for series:

`series::new`	Creates a linear combination of series.

The first parameter of `series::new` must be 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([0.1, 0.9], aapl, msft);
-- The above code is equivalent to this:
0.1 * aapl + 0.9 * msft
```

If we add another item to the vector, it will act as an independent term:

```
series([0.5, 0.1, 0.9], aapl, msft);
-- The above code is equivalent to this:
0.5 + 0.1 * aapl + 0.9 * msft
```

Series properties

These properties are applied to instances of series:

`acf`	The AutoCorrelation Function.
`amax`	Gets the maximum of the absolute values.
`amin`	Gets the minimum of the absolute values.
`count`	Gets the number of values in the series.
`fft`	Gets the Fast Fourier Transform of the values.
`first`	Gets the first point in the series (the oldest one).

`fit`	Gets a vector with two coefficients for a linear fit.
`kurt`	Get the kurtosis.
`kurtp`	Get the kurtosis of the population.
`last`	Gets the last point in the series (the newest one).
`linearFit`	Gets a line fitting the original series.
`logs`	Gets the logarithmic returns.
`max`	Get the maximum value from the series.
`mean`	Gets the average of the values.
`min`	Get the minimum value from the series.
`movingRet`	Gets the moving monthly/yearly return.
`ncdf`	Gets the percentile of the last value.
`perc`	Gets the percentiles of the series.
`random`	Creates a random series from a normal distribution.
`rets`	Gets the linear returns.
`skew`	Gets the skewness.
`skewp`	Gets the skewness of the population.
`stats`	Gets all statistics in one call.
`std`	Gets the standard deviation.
`stdp`	Gets the standard deviation of the population.
`sum`	Gets the sum of all values.
`type`	Gets the type of the series.
`var`	Gets the variance.
`varp`	Gets the variance of the population.
`values`	Gets the underlying vector of values.

Series methods

These are the methods supported by time series:

`all`	Checks if all items satisfy a lambda predicate.
`any`	Checks if exists an item satisfying a lambda predicate.
`ar`	Calculates the autoregression coefficients for a given order.
`arModel`	Creates a full AR(p) model.
`autocorr`	Gets the autocorrelation given a lag.
`corr`	Gets the correlation with a series given as a parameter.
`correlogram`	Gets all autocorrelations up to a given lag.
`cov`	Gets the covariance with another given series.
`ewma`	Calculates an Exponentially Weighted Moving Average.
`filter`	Filters points by values or dates.
`indexOf`	Returns the index where a value is stored.
`linear`	Gets the regression coefficients given a list of series.
`linearModel`	Creates a full linear model given a list of series.
`ma`	Calculates the moving average coefficients for a given order.
`maModel`	Creates a full MA(q) model.
`map`	Pointwise transformation of the series with a lambda.
`movingAvg`	Calculates a Simple Moving Average.
`movingNcdf`	Calculates a Moving Normal Percentile.
`movingStd`	Calculates a Moving Standard Deviation.
`ncdf`	Gets the normal percentile for a given value.
`stats`	Gets monthly statistics for a given date.
`zip`	Combines two series using a lambda function.

Operators

These operators can be used with time series:

`+`	Adds two series, or a series and a scalar.
`-`	Subtracts two series, or a series and a scalar. Also works as the unary negation.
`*`	Multiplies a series and a scalar for scaling values.
`/`	Divides a series by a scalar.
`.*`	Pointwise series multiplication.
`./`	Pointwise series division.

Indexing and slicing

Points in a series can be accessed 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 range operator (`..`), 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, as usual. 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 like the one of vectors:

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

Vectors

AUSTRA PROVIDES DOUBLE-PRECISION vectors, identified by the class `vec`, complex double-precision vectors, `cvec`, and vectors of integer, `ivec`. All these data types are implemented using dense storage.

Real vectors

A vector is constructed by listing its components inside brackets:

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

Commas are mandatory for separating items, and white space and line feeds are always ignored.

Bracket lists can be also used to concatenate the content of several vectors, and you can add scalars to the mix:

```
let v1=[1, 2], v2=[3, 4];
-- Returns a vector with 4 items.
[v1, v2];
-- This also is accepted:
[[1, 2], v2];
-- Scalars can also be added.
[0, v1, pi, v2, tau];
```

Class methods

Vectors can also be created using these class methods:

`vec::new`	Overloaded constructor.
`vec::ones`	Creates a vector filled with ones.
`vec::random`	Creates a vector with random values from a uniform distribution.
`vec::nrandom`	Creates a vector with random values from a normal standard distribution.

These are the overloads supported by `vec::new`:

```
-- Creates a vector with 10 items, all of them zeros.
vec::new(10);
-- Remember that ::new can be omitted!
vec(10);
```

```
-- Creates a vector like [1 2 3 4 5 6 7 8 9 10]
vec(10, i => i + 1)
```

The last example shows how to create a vector using a lambda function parameter. This is a more sophisticated example of using a lambda to initialize items in a vector:

```
-- Mimics a periodic function.
vec(1024, i => sin(i*pi/512) + 0.8*cos(i*pi/256))
```

`vec::new` can also be used to create a linear combination of vectors:

```
vec([0.5, 0.1, 0.7, 0.2], v1, v2, v3)
```

The first parameter contains weights, and the remaining parameters are the vectors that will be linearly combined. If there is an extra value in the weights, as in the example, it is used as an independent term. 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;
scalar1 * vector1 + scalar2 * vector2;
```

All these expressions are reduced to calls to one of the overloads of either `MultiplyAdd` or `Combine2`. These methods are internally optimized to use a single temporary buffer, instead of the two buffers of a naïve implementation, and both use FMA fused operations when available. Of course, the method underlying the above presented `vec::new` constructor is even better optimized and runs several faster than even the optimized versions of lineal composition.

Vector properties

Properties and methods or vectors are very similar to the ones from series. As a rule, almost all code in series that do not need to take dates into account, is implemented via the corresponding vector code, which is heavily optimized, and hardware accelerated. These are the properties supported by a vector instance:

`abs`	Gets a new vector with absolute values.
`acf`	The AutoCorrelation Function.
`amax`	Gets the maximum of the absolute values.
`amin`	Gets the minimum of the absolute values.

`distinct`	Gets a new vector with the unique values from the original one.
`fft`	Gets the Fast Fourier Transform of the values.
`first`	Gets the first item in the vector.
`last`	Gets the last item in the vector.
`length`	Gets the number of values in the vector.
`max`	Get the maximum value from the vector.
`mean`	Gets the average of the values in the vector.
`min`	Get the minimum value from the vector.
`norm`	Gets the Pythagorean norm of the vector.
`plot`	Shows the vector in a chart.
`prod`	Multiplies all items in the vector.
`reverse`	Creates a new vector with items in reverse order.
`sort`	Gets a new vector with its items sorted.
`sortDescending`	Gets a new vector with its items sorted in descending order.
`sqr`	Gets the scalar product of the vector with itself.
`sqrt`	Gets a new vector with the square root of each item.
`stats`	Gets all statistics in one call.
`sum`	Gets the sum of all values.

Vector methods

These are the methods supported by a vector instance:

`all`	Checks if all items satisfy a lambda predicate.
`any`	Checks if exists an item satisfying a lambda predicate.
`ar`	Gets the autoregression coefficients for a given `p`.
`arModel`	Creates a full AR(p) model.

`autocorr`	Gets the autocorrelation given a lag.
`correlogram`	Gets all autocorrelations up to a given lag.
`filter`	Filters items by value.
`find`	Like `filter` but returns a sequence with the indexes.
`indexOf`	Returns the first index where a value is stored.
`linear`	Gets the regression coefficients given a list of vectors.
`linearModel`	Creates a full linear model from a list of vectors.
`ma`	Estimates coefficients for an MA(q) model.
`maModel`	Creates a full MA(q) model.
`map`	Pointwise transformation of the items in a vector.
`reduce`	Reduces all items in a vector to a single value.
`zip`	Combines two vectors using a lambda function.

Vector operators

Real-valued vectors supports the basic repertoire of operators:

`+`	Adds two vectors, or a vector and a scalar.
`-`	Subtracts two vectors, or a vector and a scalar. Also works as the unary negation.
`*`	Multiplying two vectors represents the inner vector product, returning a number. A vector multiplied by a scalar is a vector scaling operation.
`/`	Divides a vector by a scalar.
`.*`	Pointwise vector multiplication.
`./`	Pointwise vector division.
`^`	Outer product for two vectors, returning a matrix.

The outer product of vectors x_i and y_j returns the matrix with components $m_{i,j} = x_i y_j$. This expression call:

```
[1,2,3]^[4,5,6]
```

computes this matrix:

```
ans ∈ ℝ(3×3)
 4   5   6
 8  10  12
12  15  18
```

Complex vectors

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

```
cvec::new([1, 2, 3, 4], [4, 3, 2, 1]);
-- ::new can be omitted.
cvec([1, 2, 3, 4], [4, 3, 2, 1])
```

These class methods are available for creating complex vectors:

`cvec::new`	Overloaded constructor (see below).
`cvec::random`	Creates a complex vector with random values from a uniform distribution.
`cvec::nrandom`	Creates a complex vector with random values from a normal standard distribution.

These are the overloads supported by `cvec::new`:

```
-- Creates a complex vector with 10 zeros.
cvec(10);
-- Creates a complex vector from one real vector.
cvec([1, 2, 3]);
-- Creates a complex vector from two real vectors.
cvec([1, 2, 3], [3, 2, 1]);
-- Creates a complex vector with a lambda function.
cvec(10, i => polar(2π*i/10));
-- The lambda function includes access to the complex vector.
cvec(100, (i, v) => polar(2π*i/10) - 0.01 * i * v{i-1})
```

Complex vector properties

These are the properties supported by a complex vector instance:

`amax`	Gets the maximum of the absolute values.
`amin`	Gets the minimum of the absolute values.

`distinct`	Gets a new vector with the unique values from the original one.
`fft`	Gets the Fast Fourier Transform of the values.
`first`	Gets the first item in the vector.
`imag`	Gets the imaginary components as a vector.
`last`	Gets the last item in the vector.
`length`	Gets the number of values in the vector.
`magnitudes`	Gets magnitudes as a vector.
`mean`	Gets the average of the values in the vector.
`norm`	Gets the Pythagorean norm of the vector.
`phases`	Gets phases as a vector.
`plot`	Shows the vector in a chart.
`prod`	Multiplies all items in the vector.
`real`	Gets the real components as a vector.
`reverse`	Creates a new vector with items in reverse order.
`sqr`	Gets the scalar product of the vector with itself.
`sum`	Gets the sum of all values.

Complex vector methods

These are the methods supported by a complex vector instance:

`all`	Checks if all items satisfy a lambda predicate.
`any`	Checks if exists an item satisfying a lambda predicate.
`filter`	Filters items by value.
`find`	Like `filter` but returns a sequence of the indexes.
`indexOf`	Returns the first index where a value is stored.
`map`	Pointwise transformation of the items in a vector.

`mapReal`	Pointwise transformation of the items in a vector. Returns a real vector.
`reduce`	Reduces all items in a vector to a single value.
`zip`	Combines two vectors using a lambda function.

Complex vector operators

Complex vectors support the same operators as real-valued operators, except for the outer product `^`. On the other hand, they add support for complex vector conjugation using a unary suffix operator:

`'`	Unary suffix operator for complex vector conjugation.

Complex vector conjugation inverts the sign of each imaginary component in the vector. The inner product of two complex vectors conjugates the second vector operand.

Integer vectors

Integer vectors are also supported, using the `ivec` class.

`ivec::new`	Overloaded constructor.
`ivec::ones`	Creates a vector filled with ones.
`ivec::random`	Creates a vector with random values from a uniform distribution.

`ivec::random` has three overloaded variants:

```
-- Ten items. Values between 0 and int.MaxValue - 1.
ivec::random(10);
-- Values between 0 and 999.
ivec::random(10, 1000);
-- Values between -10 and 9.
ivec::random(-10, 10)
```

Integer vector properties

Integer vectors support these properties:

`abs`	Gets a new vector with absolute values.
`distinct`	Gets a new vector with the unique values from the original one.
`first`	Gets the first item in the vector.

`last`	Gets the last item in the vector.
`length`	Gets the number of values in the vector.
`max`	Gets the maximum value from the vector.
`min`	Gets the minimum values from the vector.
`prod`	Multiplies all items in the vector.
`reverse`	Creates a new vector with items in reverse order.
`sort`	Sorts the vector in ascending order.
`sortDesc`	Sorts the vector in descending order.
`stats`	Gets all statistics in one call.
`sum`	Gets the sum of all values.
`toVector`	Converts the integer vector into a double vector.

Integer vector methods

These are the methods for integer vectors:

`all`	Checks if all items satisfy a lambda predicate.
`any`	Checks if exists an item satisfying a lambda predicate.
`filter`	Filters items by value.
`map`	Pointwise transformation of the items in another integer vector.
`mapReal`	Pointwise transformation of the items into a real vector.
`reduce`	Reduces all items in a vector to a single integer value.
`zip`	Combines two integer vectors using a lambda function.

Indexing and slicing

Individual values from vectors are accessed using its position, starting from zero, inside brackets:

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

A segment or slice 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]
```

These equalities hold:

```
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. This expression creates a vector holding the first 30 Fibonacci numbers:

```
vec(30, (i, v) => max(1, v{i-1} + v{i-2}))
```

Sequences

SEQUENCES PROVIDES MOST of the operations from real and complex vectors but avoiding the storage. Sequences are like enumerable types in C# with LINQ and are a requisite for any functional language.

AUSTRA supports three kinds of sequences: `seq`, for real valued sequences, `cseq` for complex ones, and `iseq`, for integer sequences.

Double-valued sequences as light vectors

Let's say we want to calculate factorials. AUSTRA is a functional language, so we don't have explicit loops. We could, however, do this with a vector:

```
vec(10, i => i + 1).prod
```

The above code works fine, but it forces the library to allocate one array of ten items. This is the alternative, using a sequence:

```
seq(2, 10).prod
```

Since the sequence's values are generated only by demand, there's no need for the internal storage.

Sequence constructors

These are the class methods for `seq`:

`seq::new`	Creates a sequence, either from a range, a range and a step, or from a vector or matrix. See examples below.
`seq::random`	Creates a sequence of random values.
`seq::nrandom`	Creates a sequence of random values, using a normal distribution.
`seq::ar`	Creates a sequence using an AutoRegressive process.
`seq::ma`	Creates a sequence using a Moving Average process.
`seq::unfold`	Generate values from a seed and a generating function.

This code fragment shows some of the available constructors for sequences:

```
seq(1, 10);          -- Numbers from 1 to 10.
seq(10, 1);          -- The inverted sequence.
seq::new(1, 10);     -- ::new was omitted before.
seq(0, 128, τ);      -- A uniform grid with 128 intervals.
seq(v);              -- A sequence from a vector.
seq([sqrt(2), e, π, τ]);
seq(v1^v2);          -- A sequence from a matrix.
seq::random(10);     -- A sequence with 10 random values.
seq::nrandom(10),    -- A sequence with 10 Gaussian random values.
seq::nrandom(10, 2) -- Ten normal samples with variance = 2.
```

Real sequences created with just a lower and an upper bound are always based on integer bounds. For instance, these two expressions represent the same real sequence:

```
seq(1, 10);          -- Numbers from 1 to 10.
seq(1.5, 10.6);      -- Numbers from 1 to 10, too.
```

The reason for this rule is that these sequences have always one unit as their step. It is easier to reason about their behaviour knowing that their boundaries always are integer values.

There are two additional class methods for generating autoregressive, AR(p), and moving average, MA(q), sequences:

```
-- An autoregressive (AR) process of order three.
seq::ar(1000, 1, [0.1, 0.05, 0.01]);
-- A moving average (MA) process of order three.
-- The first term in the vector is the model's mean.
seq::ma(1000, 1, [0, 0.1, 0.05, 0.01])
```

You can materialize the content of a sequence as a vector using the `toVector` property:

```
seq::random(10).toVector
```

The unfold sequence generator

Another class method for creating sequences is `seq::unfold`, which has three variants:

```
-- Powers of 2, from 2 to 1024.
seq::unfold(10, 2, x => 2x);
-- Maclaurin series for exp(1).
seq::unfold(100000, 1, (n, x) => x / (n + 1)).sum + 1;
-- Real-valued Fibonacci sequence.
seq::unfold(50, 1, 1, (x, y) => x + y);
```

The **`unfold`** sequence generator is important for the language since it can express iterative behaviour in a language with no explicit loops and conditionals. The

alternative to iteration is recursion, which is also provided by AUSTRA, but a recursive function is almost always more expensive, even in the presence of tail optimizations.

Let us consider, for instance, how we would write a function for the greatest common divisor. When learning about function definitions, we saw that we could define a recursive function for this task:

```
-- The recursive version
def gcd1:"Recursive GCD"(a, b: int): int =
    let rm = a % b in
        iff(rm = 0, b, mcd(b, rm))
```

This is a fine recursive definition that even is amenable to tail recursion optimization. Now compare with the iterative alternative:

```
-- The iterative version
def gcd2:"Iterative GCD"(a, b: int): int =
    iseq::unfold(10000, a, b, (x, y) => x%y).while(x => x > 0).last
```

Even though it looks more verbose, the second definition is almost twice faster than the recursive definition. Note, however, that we have used the class `iseq` instead of `seq`, so the property `last` directly returned an integer value. We could have kept `seq` by simply adding `toInt` after calling `last`:

```
def gcd3:"Iterative GCD"(a, b: int): int =
    seq::unfold(10000, a, b, (x, y) => x%y).while(x => x >
0).last.toInt
```

Still, the advantage of the iterative definition will be huge compared to the recursive definition.

Note

Two things to have into account:

- Have we not included the while method, every call to that unfold sequence would end with failure as soon as the algorithm would have tried to divide by zero. The while method avoids that problem and gives as the value we need.
- The unfold generator requires a first parameter stating how many values to create. We could have devised a variant of unfold for creating infinite length sequence. I have preferred, however, to put the burden of picking a high enough value on your shoulders. It is security against convenience.

Methods and properties

These are the methods that can be used with a sequence:

`all`	Checks if all items in the sequence satisfy a predicate.

`any`	Checks if there is an item in the sequence satisfying a predicate.
`ar`	Estimates coefficients for an AR(p) model.
`arModel`	Creates a full AR(p) model.
`filter`	Returns items of the original sequence satisfying a predicate.
`ma`	Estimates coefficients for an MA(q) model.
`maModel`	Creates a full MA(q) model.
`map`	Transforms items with the help of a lambda function.
`reduce`	Conflates all values in a sequence using a lambda.
`until`	Returns a prefix of a sequence until a value satisfying a predicate is found.
`while`	Returns a prefix of a sequence while values satisfy a predicate.
`zip`	Combines two sequences using a lambda function.

As seen before, the `while` method excludes from its returning sequence the first value satisfying its predicate. On the contrary, `until` includes that very value in the returning sequence.

Properties are methods without parameters, that can be called without parentheses. These are the supported properties:

`distinct`	Select unique values in the sequence, with no predefined order.
`fft`	Calculates a Fast Fourier Transform.
`first`	Gets the first term of the sequence.
`last`	Gets the last term of the sequence.
`length`	Gets the number of elements in the sequence.
`max`	Get the maximum value in the sequence.
`min`	Get the minimum value in the sequence.
`plot`	Plots the sequence.
`prod`	Multiplies all values in the sequence.
`sort`	Sorts values in ascending order.

`sortDesc`	Sorts values in descending order.
`stats`	Gets all statistic moments of the sequence.
`sum`	Sums all values in the sequence.
`toVector`	Materializes the sequence into a vector.

Sequence operators

Sequence's operators mimics most of vector's operators.

```
seq(1, 10) * seq(10, 1)  -- The dot product.
```

For instance, simple operators can be used to change the underlying distribution of a random sequence.

```
seq::random(100) * 2 - 1;
-- Check the moments of the above distribution.
(seq::random(100) * 2 - 1).stats
```

Note

Unary operators for sequences could, in theory, be implemented using `map`, and binary operators can also be written using `zip`.

However, in most cases, having an explicit operator results in a faster implementation. It is most evident for sequences backed by a vector, but it also happens for other kinds of sequences. For instance, when a range or grid sequence is negated, you can implement the result using another range or grid sequence.

Integer sequences

Integer sequences are represented by the `iseq` class.

Class methods

These are the class methods supported by `iseq`:

`iseq::new`	Creates a sequence, either from a range, a range and a step, or from an integer vector.
`iseq::random`	Creates a sequence of random integers. You can pass an upper bound, or an interval for values.
`iseq::unfold`	Like `seq::unfold`, but with integer arguments.

Methods and properties

These properties can be used with integer sequences:

`distinct`	Select unique values in the sequence, with no predefined order.
`first`	Gets the first term of the sequence.
`last`	Gets the last term of the sequence.
`length`	Gets the number of elements in the sequence.
`max`	Get the maximum value in the sequence.
`min`	Get the minimum value in the sequence.
`plot`	Plots the sequence.
`prod`	Multiplies all values in the sequence.
`sort`	Sorts values in ascending order.
`sortDesc`	Sorts values in descending order.
`stats`	Gets all statistic moments of the sequence.
`sum`	Sums all values in the sequence.
`toVector`	Materializes the sequence into a vector.

These are the available methods:

`all`	Checks if all items in the sequence satisfy a predicate.
`any`	Checks if there is an item in the sequence satisfying a predicate.
`filter`	Returns items of the original sequence satisfying a predicate.
`map`	Transforms items with the help of a lambda function.
`mapReal`	Transforms items with the help of a lambda function.
`reduce`	Conflates all values in a sequence using a lambda.
`until`	Returns a prefix of a sequence until a value satisfying a predicate is found.
`while`	Returns a prefix of a sequence while values satisfy a predicate.
`zip`	Combines two sequences using a lambda function.

This example shows how to calculate the Collatz sequence using integer sequences:

```
let collatz(n: int) =
    iseq::unfold(1000000, n, x => iff(x % 2 = 0, x / 2, 3x + 1))
    .until(x => x = 1);
collatz(137)
```

Though the generator is created with a big enough upper limit, the sequence stops when a `1` is generated. The `until` method can also be written this way:

```
let collatz(n: int) =
    iseq::unfold(1000000, n, x => iff(x % 2 = 0, x / 2, 3x + 1))
    .until(1)
```

Complex sequences

Complex sequences can also be used, with the `cseq` class.

Class methods

These are the class methods supported by `cseq`:

`cseq::new`	Creates a sequence, either from a complex interval, or from a complex vector.
`cseq::random`	Creates a sequence of random values from a uniform distribution.
`cseq::nrandom`	Creates a sequence of random values with a standard normal distribution.
`cseq::unfold`	Like `seq::unfold`, but with complex arguments.

Methods and properties

These properties can be used with complex sequences:

`distinct`	Select unique values in the sequence, with no predefined order.
`first`	Gets the first term of the sequence.
`last`	Gets the last term of the sequence.
`length`	Gets the number of elements in the sequence.
`plot`	Plots the sequence.
`prod`	Multiplies all values in the sequence.
`sum`	Sums all values in the sequence.

`toVector`	Materializes the sequence into a complex vector.

And these are the available methods:

`all`	Checks if all items in the sequence satisfy a predicate.
`any`	Checks if there is an item in the sequence satisfying a predicate.
`filter`	Returns items of the original sequence satisfying a predicate.
`map`	Transforms items with the help of a lambda function.
`mapReal`	Transforms items with the help of a lambda function.
`reduce`	Conflates all values in a sequence using a lambda.
`until`	Returns a prefix of a sequence until a value satisfying a predicate is found.
`while`	Returns a prefix of a sequence while values satisfy a predicate.
`zip`	Combines two sequences using a lambda function.

Delayed execution

Sequences are modelled after .NET LINQ enumerable interfaces, and so many other functional libraries. One of the most interesting features of these libraries is *delayed execution*.

Applying a method or an operator on a sequence does not means that it will automatically scan the sequence values. Let's start with a simple example:

```
-seq(1, 1000)
```

The above code first creates a sequence that will enumerate numbers from 1 to 1000. Creating the sequence means creating a small instance of an internal class that can be called later to yield the values in the sequence. The unary minus, however, take that sequence generator and returns another generator that yields values in descending order, from the interval *[-10, -1]*. It does not force yet the sequence enumeration. Enumeration takes place as the last operation, as you hit F5 in the AUSTRA desktop, as the application needs to print the values created by the expression. The same would happen with this expression, that plots the sequence as a series:

```
(-seq(1, 1000)).plot
```

It is the `plot` method the trigger which starts the internal loop for generating all the values. You could even intercalate another method call before the plot, without triggering enumeration:

```
-- Sort the negated values in ascending order.
(-seq(1, 1000)).sort.plot;
-- Square values, select multiples of three and sort descending.
seq(1, 1000).map(x => x^2).filter(x => x % 3 = 0).sortDesc.plot;
-- Methods like sum, prod, any,or first can also trigger evaluation.
seq(1, 100).filter(x => x % 2 = 0).map(x => x^2).sum
```

Matrices

AUSTRA MATRICES ARE represented by the `matrix` class. They are implemented as row-first, double precision dense matrices.

The AUSTRA `matrix` class is based on three different C# structures: `Matrix`, `LMatrix`, and `RMatrix`. The compiler takes automatically care of any conversions when needed.

Matrix construction

A matrix can be constructed by enclosing its components inside brackets:

```
[1, 2, 3; 2, 3, 4; 3, 4, 5]
```

Rows must be separated by semicolons (`;`), and items in a row must be separated by commas. This syntax does not allows writing a matrix with only one row, since the compiler would not be able to tell it from a vector. A workaround is writing a matrix with only one column, and transposing it:

```
[1; 2; 3; 4]'
```

You can also create a new matrix by concatenating two existing matrices, or a matrix and a vector. You can use either vertical or horizontal concatenation:

```
let m = [1,2;3,4], v=[1, 1];
-- Horizontal concatenation (2x4 matrix).
[m, m];
-- Horizontal concatenation (2x6 matrix).
[m, m, m];
-- Horizontal concatenation (2x3 matrix).
[m, v];
[v, m];
-- Vertical concatenation (4x2 matrix).
[m; m];
-- Vertical concatenation (6x2 matrix).
[m; m; m];
-- Vertical concatenation (3x2 matrix).
-- The vector is handled as a row vector.
[m; v];
[v; m];
```

Class methods

These class methods are available for creating matrices:

`matrix::new`	Overloaded constructor (see below).
`matrix::rows`	Creates a matrix given its rows as vectors.
`matrix::cols`	Creates a matrix given its cols as vectors.
`matrix::eye`	Creates an identity matrix given its size.
`matrix::diag`	Creates a diagonal matrix given the diagonal as a vector.
`matrix::random`	Creates a matrix with random values from a uniform distribution.
`matrix::nrandom`	Creates a matrix with random values from a normal standard distribution.
`matrix::lrandom`	Creates a lower-triangular matrix with random values from a uniform distribution.
`matrix::lnrandom`	Creates a lower-triangular matrix with random values from a standard normal distribution.
`matrix::cov`	Creates a covariance matrix given a list of series.
`matrix::corr`	Creates a correlation matrix given a list of series.

Methods and properties

These are the properties available for matrices:

`amax`	Gets the absolute maximum.
`amin`	Gets the absolute minimum.
`chol`	Calculates the Cholesky factorization.
`cols`	Gets the number of columns.
`det`	Calculates the determinant.
`diag`	Gets the main diagonal as a vector.
`evd`	Calculates the EigenValues Decomposition.
`inverse`	Gets the inverse of this matrix.
`isSymmetric`	Verifies if the matrix is a symmetric one.

`max`	Gets the maximum value from the cells.
`min`	Gets the minimum value from the cells.
`rows`	Gets the number of rows.
`trace`	Gets the sum of the main diagonal.
`stats`	Returns statistics on cells.

And these are the supported methods:

`all`	Checks if all cells satisfy a lambda predicate.
`any`	Checks if exists a cell satisfying a lambda predicate.
`getCol`	Gets a column by its index.
`getRow`	Gets a row by its index.
`map`	Creates a new matrix with transformed cells.

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. It is also used as a unary operator.
`*`	Matrix * matrix = matrix multiplication. Matrix * number = matrix scale. Matrix * vector = vector transformation. Vector * matrix = vector is transposed and then transformed.
`.*`	Pointwise multiplication of two matrices.
`./`	Pointwise quotient of two matrices.
`/`	Divides a matrix by a scalar, but also divides either a vector or a matrix by a matrix, for solving linear equations.
`'`	Unary suffix operator for matrix transpose.

These examples show how to solve linear equations for a vector, using division by a matrix:

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

Solving equations for a matrix is also possible:

```
let m = matrix::random(5) + 1;
matrix::eye(5) / m - m.inverse
```

Internally, the LU factorization of the matrix is used for equation solving, for the general case. When the matrix at the left is a triangular matrix, a most efficient algorithm is used.

Indexing and slicing

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

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

All indexes start 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,];
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:

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

List comprehensions

A LIST COMPREHENSION IS a syntactic sugar construct for filtering and mapping sequences, vectors, and series. They simplify writing lambda functions for methods, and they are easier to read and understand.

Syntax

Suppose you need to write a formula like this one:

```
seq(1, 100).filter(x => x % 2 = 1).map(x => x^2)
```

This code is not a candidate for the Turing Award: it takes the squares of all even numbers between 0 and 100. You had to type two lambda functions, including arrows and lambda parameters, and you also had to explicitly mention the `filter` and the `map` methods, including the parentheses enclosing their arguments.

This alternative expression does the same, is shorter to type and easier to read:

```
[x <- seq(1, 100) : x % 2 = 1 => x^2]
```

Since the source of all numbers is a range sequence, you could also use a simpler expression for the range:

```
[x <- 1..100 : x % 2 = 1 => x^2]
```

The syntax for this construct can be summarized like this:

```
[identifier <- generator : filter => mapping]
```

Both `filter` and `mapping` are optional:

```
-- This expression...
[x <- 1..100];
-- ... is equivalent to this one:
seq(1, 100)
```

Types in list comprehensions

The type assigned to the whole list comprehension expression is the same of its generator. You can keep applying methods or operators to the result:

```
[x <- 1..100 : x % 2 = 1 => x^2].sortDesc;
[x <- 1..100] .* ([x <- 1..100] + 1)
```

Special care is needed when the generator is a time series, because the identifier in the head of the list comprehension is typed as `double` in the mapping section, but it is a `Point<Date>` in the filter section:

```
let mean = msft.mean in
  [x <- msft : x.date >= jan2015 => x - mean]
```

Generators

As we have seen, range expressions can be used as generators. We support four variants of range expressions inside list comprehensions:

```
-- Equivalent to iseq(1, 100)
[x <- 1..100];
-- Equivalent to seq(1, 100)
[x <- 1.0..100.0];
-- Even integers from 0 to 100.
[x <- 0..2..100];
-- The same as seq(0, 1024, 2 * pi)).
[x <- 0..1024..2pi];
```

In the last example, only the upper bound is real, so the compiler handles the generator as a real sequence.

The parameter identifier and the membership operator can also be drop, and the above examples simplify this way:

```
[1..100];
[1.0..100.0];
[0..2..100];
[0..1024..2pi];
```

We have mostly used constants for the range generators so far but, of course, each part of the generator could be an expression:

```
[x <- pi - 1..32 * 32..sqrt(200)];
```

Quantifiers in list comprehensions

Logical quantifiers can be used at the beginning of a list comprehension. The allowed quantifiers are `all` and `any`, as the corresponding methods in vectors and sequences. They are no keywords, but used at the beginning of a list comprehension, they are considered *contextual keywords* for syntax highlighting.

This is a very simple example of a quantifier in a list comprehension expression and its equivalent form using methods:

```
[any x <- 10..100 : x * x = x + x];
iseq(10, 100).any(x => x * x = x + x)
```

Both expressions are compiled as Boolean expressions. Note that a qualified list comprehension does not allow a mapping section.

The quantified list comprehension is marginally shorter than a call to `any` or `all`. Why, then, do we bother supporting this syntax? The reason is that we can embed a qualified predicate inside a normal list comprehension:

```
-- Find all prime numbers between 2 and 100:
[x <- 2..100 : all div <- 2 .. x - 1 : x % div != 0];
-- Equivalent, but longer:
iseq(2, 100).filter(x => iseq(2, x - 1).all(div => x % div != 0))
```

We need no inner brackets inside the main list comprehension, since it is evident how the qualified condition is nested. We could even add a mapping at the end of the comprehension to transform the calculated prime numbers, if required.

If we use regular lambdas, we will be nesting a lambda definition inside another. The generated code for the list comprehension also uses nested lambdas, but with an easier to understand syntax. The inner lambda is "capturing" the parameter of the outer lambda, so we must be careful naming local variables.

Splines

SPLINES ARE PIECEWISE defined functions, using cubic polynomials, for interpolating or smoothing curves. Austra can create splines for time series, using dates as arguments, or for any pair of vectors containing abscissas and coordinates, respectively. There is also a shortcut for creating this second kind of splines given a grid on an interval and an arbitrary function.

Creating splines

All spline kinds are created using overloaded variants of the same class method:

`spline::new`	Creates a spline either from a series, a couple of vectors, or a grid and a lambda function.

This example shows how to create and use a spline based on a time series:

```
let s = spline(appl) in
    s[appl.last.date - 15]
```

The example creates a spline based on the series values, and then the spline is used to interpolate the value fifteen days before the last date stored in the series.

Note

Interpolation for daily series does not have much sense from the AUSTRA language since our `date` type does not include a time.

Splines can also be used to interpolate existing data and functions:

```
-- Use a function over a uniform grid.
let s1 = spline(0, τ, 1024, cos);
s1[π/4] - sqrt(0.5);
s1.derivative(π/4);
-- Use two arbitrary vectors with the same length.
let s2 = spline([1, 3, 4, 5], [0, 1, 0.8, 0]);
s2[2]
```

Indexers, methods, and properties

All splines have these four properties:

`area`	The total area below the spline.

`first`	The lower bound for the abscissas. It is a date for splines based on series, and a double value, otherwise.
`last`	The upper bound for the abscissas. It is a date for splines based on series, and a double value, otherwise.
`length`	Gets the number of polynomials in the spline.

For instance, we can use it to approximate the area below a normal distribution:

```
-- The integral over a reasonable interval.
spline(-10, 10, 10000, x => exp(-x²)).area;
-- The expected result.
sqrt(π)
```

Note

When `area` is used on a series-based spline, dates are automatically interpreted as real values, so a day is equal to the unit value.

These are the methods implemented by splines:

`derivative`	Calculates the smoothed derivative at a given point of the spline range.
`poly`	Gets the cubic polynomial at a given index in the spline.

Index