**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.