Symbolic C# framework for vector and matrix operations

Sergey L. Gladkiy

Introduction

Vector and matrix data appeared in many modern applications, such as statistical, economical, spreadsheets, databases, analytics and forecasting tools, image and video processing, math and physical modeling, artificial intelligence and so on. The data can be processed by the application with the predefined, built-in formulae. But sometimes, there are the processing cases with user defined formulae. For an example, the user can define his own formula for a signal processing filter, and so, this formula cannot be built in the application. For such special cases the application must include a tool for user defined formula processing – so called 'calculator' or 'formula parser'.

This article describes the capabilities of the symbolic framework **ANALYTICS** for working with vector and matrix data. The framework is written entirely in C# and has some unique features those allow easily introduce new operations for vector and matrix data and integrate the framework with any application.

Background

Vector and matrix operations play an important role in many data processing applications. For an example, the simplest case is calculating the **mean value** of some data set. The formula for mean value, described via vector data, is the following:

$$\mu = \frac{1}{N}\sum_{i=1}^{N} B_i$$

where ***B*** is the vector and ***N*** is the number of its components. It is obvious, that an application, that deals with statistical data processing must include built-in algorithm for mean value evaluation, because this is one of the base formulae.

But let us imagine, that some user invented new formula for statistical processing, for an example, new probability distribution function. Then, there is no built-in formula for the distribution modeling in the software. The advanced processing applications use special tools for handling this case. The tools allow user to input his own formula for data processing.

There are many frameworks for realizing such tools in data processing applications. The frameworks commonly allow math formula evaluation, including real numbers, functions, and math operators. In rare cases they allow evaluating formula with complex numbers or vector data. And if they do, it can be not enough to realize a vector formula evaluation tool, because different applications have different built-in data types, and the framework must allow define operations for any data type.

Let us state here the minimal requirements for symbolic framework with vector and matrix operations:

- Parsing math formula of any complexity, independently on the data types.
- Support vector and matrix notations, including indexing and special operators.
- Evaluating math formula of any complexity for arbitrary data types, including data compatibility control.
- Introducing new functions and overloading operators for any external types without modifying core framework algorithms.
- Integrating the framework into any target application with minimal code writing.

To satisfy these requirements, the symbolic framework should have specially designed, strongly structured, abstract class hierarchy. The **ANALYTICS** symbolic framework written in C# and specially designed for satisfying the

requirements, including supporting vector and matrix operations (http://sergey-l-gladkiy.narod.ru/index/analytics/0-13).

Realization

As stated above, the symbolic framework must follow the special class hierarchy. The main goal of the classes is to evaluate any math formula, independently on the data types it contains. This can be realized by introducing abstract classes for various math concepts (operators, functions and so on) and then inheriting specific classes for concrete concepts and data types. Let us consider the realization of the idea for the math operator.

Syntactically an operator is a symbol standing for some mathematical operation. For example, in the expression 'x+y' the operator '+' stands for the addition operation. From the functional point of view, an operator implements some action with data value(s), called operand(s), and returns the result of the operation.

All operators can be divided into many categories. The most common used are unary and binary operators. An unary operator has one operand and can be prefix (stands in front of its operand) and postfix (stands behind its operand). An example of an unary prefix operator is the negation operator ('-x', '-' is the operator, 'x' is the operand). An example of an unary postfix operator is the factorial operator ('n!', '!' is the operator, 'n' is the operand). Binary operators have two operands and commonly stand between them. An example of a binary operator is the addition operator ('x+y', '+' is the operator, 'x' and 'y' are the operands).

Taking into account the definition of math operator, we can introduce the following abstract class for the concept:

```
/// <summary>
/// Base abstract class for all operators
/// </summary>
public abstract class Operator
{
    protected abstract OperatorType GetOperatorType();
    protected abstract Type GetReturnType();

    /// <summary>
    /// Does operation
    /// </summary>
    protected abstract object DoOperation(object[] operands);

    /// <summary>
    /// Type of operator
    /// </summary>
    public OperatorType Type
    {
        get
        {
            return GetOperatorType();
        }
    }

    /// <summary>
    /// Type of result
    /// </summary>
    public Type ReturnType
    {
        get
        {
            return GetReturnType();
```

```
        }
    }

    /// <summary>
    /// Arity (depends on Operator Type)
    /// </summary>
    public OperatorArity Arity
    {
        get
        {
            return Utilities.GetOperatorArity(Type);
        }
    }

    /// <summary>
    /// Position (depends on Operator Type)
    /// </summary>
    public OperatorPosition Position
    {
        get
        {
            return Utilities.GetOperatorPosition(Type);
        }
    }

    /// <summary>
    /// Calculates operation result
    /// </summary>
    public object Calculate(object[] operands)
    {
        object result = DoOperation(operands);
        return result;
    }
}
```

The main properties of the class are: ***Type*** – the operator type (add, subtract, multiply and so on); ***ReturnType*** – the type of the operation (used for controlling the operations type compatibility). The main method of the class is ***Calculate*** – operation with operand values which can be overridden for specific operator classes.
And then we can introduce specific classes for unary and binary operators:

```
/// <summary>
/// Base abstract class for ALL Unary operators
/// </summary>
public abstract class UnaryOperator : Operator
{
    protected abstract Type GetOperandType();
    protected abstract object Operation(object operand);

    /// <summary>
    /// Does operation with one operand
    /// </summary>
    protected override sealed object DoOperation(object[] operands)
    {
        return Operation(operands[0]);
    }

    /// <summary>
```

```
    /// Type of operand
    /// </summary>
    public Type OperandType
    {
        get
        {
            return GetOperandType();
        }
    }
}

/// <summary>
/// Base abstract class for ALL Binary operators
/// </summary>
public abstract class BinaryOperator : Operator
{
    protected abstract Type GetOperand1Type();
    protected abstract Type GetOperand2Type();

    /// <summary>
    /// Operation with two operands
    /// </summary>
    protected abstract object Operation(object operand1, object operand2);

    /// <summary>
    /// Does operation with two operands
    /// </summary>
    protected override sealed object DoOperation(object[] operands)
    {
        return Operation(operands[0], operands[1]);
    }

    /// <summary>
    /// Type of 1st operand
    /// </summary>
    public Type Operand1Type
    {
        get
        {
            return GetOperand1Type();
        }
    }

    /// <summary>
    /// Type of 2nd operand
    /// </summary>
    public Type Operand2Type
    {
        get
        {
            return GetOperand2Type();
        }
    }
}
```

The class for unary operator introduces new property ***OperandType***, which used again for providing the type safety of the operations. And also it overrides the ***DoOperation*** method, taking into account that the operator has only one operand. Analogously, the class for binary operator provides interface for two operands.

Thus, the main purpose of the operator classes is to provide evaluation functions (the ***Calculate*** method) to the internal calculation engine (not described here because it is out of the article's scope). The calculation engine will be able to find (through the .NET reflection mechanisms) the appropriate operator class for various expressions. For an example, if in the expression 'x+y', the 'x' is a complex variable and the 'y' is the real variable, the calculation engine will search the class of binary operator with operator type '***Add***' and with operand types '***Complex***' and '***double***' respectively. If there is an appropriate operator class – it will be used for performing the operation, else – the engine will notify (through the exception mechanism) that the operation cannot be done. This technology of operator overloading allows introducing operations for any data types via external plugins (.NET assemblies) without modifying the core algorithms of the **ANALYTICS** framework.

Another thing to do is providing simplicity of the operations introducing. Consider the class of binary operator. For introducing the operation for some operand types we must override getters for the ***ReturnType***, ***Operand1Type***, ***Operand2Type*** and ***Type*** properties, as the ***Operation*** method. Despite of the simple provided interface we can still improve it, using generics. Let us consider this for binary operators:

```
/// <summary>
/// Abstract Generic Binary operator.
/// </summary>
public abstract class GenericBinaryOperator<Operand1Type, Operand2Type, ReturnType> : BinaryOperator
{
    /// <summary>
    /// Typed Binary Operator Operation.
    /// </summary>
    protected abstract ReturnType TypedOperation(Operand1Type operand1, Operand2Type operand2);

    protected override sealed Type GetOperand1Type()
    {
        return typeof(Operand1Type);
    }

    protected override sealed Type GetOperand2Type()
    {
        return typeof(Operand2Type);
    }

    protected override sealed Type GetReturnType()
    {
        return typeof(ReturnType);
    }

    /// <summary>
    /// Overrides Binary Operator Operation replacing it
    /// by typed one with automatic operands cast.
    /// </summary>
    protected override sealed object Operation(object operand1, object operand2)
    {
        Operand1Type x1 = (Operand1Type)operand1;
        Operand2Type x2 = (Operand2Type)operand2;

        return TypedOperation(x1, x2);
    }
}
```

The generic binary operator overrides the most of the methods, using generic parameter types. It also introduces the ***TypedOperation*** method, which receives the arguments of the specified types. And final improvement is introducing generic classes for specific operations – addition, subtraction and so on. The class for addition operation is:

```
/// <summary>
/// Base Generic class for Add (+) operators
/// </summary>
public abstract class GenericAddOperator<Operand1Type, Operand2Type, ReturnType> :
GenericBinaryOperator<Operand1Type, Operand2Type, ReturnType>
{
    protected override sealed OperatorType GetOperatorType()
    {
        return OperatorType.Add;
    }
}
```

The introduced operator class hierarchy seems very complicated. But from the point of view of a developer, who uses the ANALYTICS framework for creating data processing application, it is very simple, because he has to use only specific generic classes, like the ***GenericAddOperator***. And there is only one method ***TypedOperation*** which must be overridden for final operator classes. For an example, the class of addition operation for two vectors (arrays of double values) is the following:

```
/// <summary>
/// (Double Array) + (Double Array) = (Double Array)
/// </summary>
public sealed class ArrayAdd : GenericAddOperator<double[], double[], double[]>
{
    protected override double[] TypedOperation(double[] operand1, double[] operand2)
    {
        return ((new Vector(operand1)) + (new Vector(operand2))).Data;
    }
}
```

where the ***Vector*** class is internal structure, performing math operations with one-dimensional double arrays.
As can be seen from the code above, introducing operations for any data types is very simple and straightforward. Following the approach, described for introducing math operators, we can provide abstractions for other elements of math expressions, like **literals** (**constants**), **variables**, **functions** (https://en.wikipedia.org/wiki/Expression_(mathematics)). Such abstraction layer totally realized in the **ANALYTICS** framework (more information here http://sergey-l-gladkiy.narod.ru/ana_docs/ANALYTICS_C_manual.en.pdf). The library also provides realization of all vector and matrix operations for real and even complex elements. The library's functionality for **logical** (Boolean) arrays and matrixes provides base functionality for statistical data processing.

Examples of using symbolic formula with vector and matrix operations

Now, let us consider some examples of using vector and matrix symbolic formula for solving several problems of data processing. The first one is calculation of statistical deviation for the set of real values. The formula for the deviation value (https://en.wikipedia.org/wiki/Standard_deviation) is the following:

$$D = \frac{1}{N}\sum_{i=1}^{N}(B_i - \mu)^2$$

where μ is the mean value $\mu = \frac{1}{N}\sum_{i=1}^{N} B_i$, N is the number of elements. The formula evaluation can be implemented with the following code:

```
Translator translator = new Translator(); // Initializing the translator instance.

// Vector data (can be received from any source - file, network...)
double[] bv = new double[] { -0.2, 0.2, -0.1, 0.25, 0.35, 0.15, -0.04, 0.01, -0.12, 0.1 };

translator.Add("B", bv); // Add vector variable.
string mf = "∑B/#B"; // Mean formula for "B" vector.
double m = (double)translator.Calculate(mf); // Calculate the mean value.
translator.Add("μ", m); // Add the variable for the mean value.
string df = "∑((B-μ)^2)/#B"; // Deviation formula for "B" vector.
double d = (double)translator.Calculate(df); // Calculate the deviation value.
```

Here the ***Translator*** class is the calculation engine, mentioned above; the '**#**' operator calculates the number of vector or matrix elements; the '**∑**' operator calculates the sum of all vector elements. The output for the evaluation of example data is:

B = (-0.2 0.2 -0.1 0.25 0.35 0.15 -0.04 0.01 -0.12 0.1)
∑B/#B = 0.06
∑((B-μ)^2)/#B = 0.02876

It should be noted here that the code and formulae are independent on the size of processing data. The evaluation result is correct for any array, which can be received from any source: user input, data file, spreadsheet, network or any other.

The next example is a little complicated problem: solving an over-determined linear equation system with the least squares approach (https://en.wikipedia.org/wiki/Linear_least_squares_(mathematics)). Let the system is set up by the '**M**' matrix and the '**A**' vector. Then the solution '**X**' is defined by the formula:

$$X = (M^T M)^{-1} M^T A,$$

where M^T – is the transposed matrix, $^{-1}$ means the inversed matrix.

The problem can be solved with the following code:

```
Translator translator = new Translator(); // Initializing the translator instance.

double[,] mv = new double[,] // Matrix 4x3.
{
    {  0.5,  -0.4,  0.33 },
    { -0.3,   0.1,  0.28 },
    { -0.4,  -0.2, -0.75 },
    {  0.44,  0.25, 1.00 },
};
translator.Add("M", mv); // Add the matrix variable.

double[] av = new double[] { 0.2, -0.1, -0.25, 0.4 }; // Vector 4.
translator.Add("A", av); // Add the vector variable.
```

```
string xf = "M'*A/(M'*M)"; // Least squares solution formula.
// Solve the system with the least squares approach.
double[] x = (double[])translator.Calculate(xf);
```

Here the apostrophe ‘’’ operator transposes the matrix; the division operation used instead of multiplying by the inversed matrix. And the output for the problem solution is:

M =
(0.5 -0.4 0.33)
(-0.3 0.1 0.28)
(-0.4 -0.2 -0.75)
(0.44 0.25 1)

A = (0.2 -0.1 -0.25 0.4)

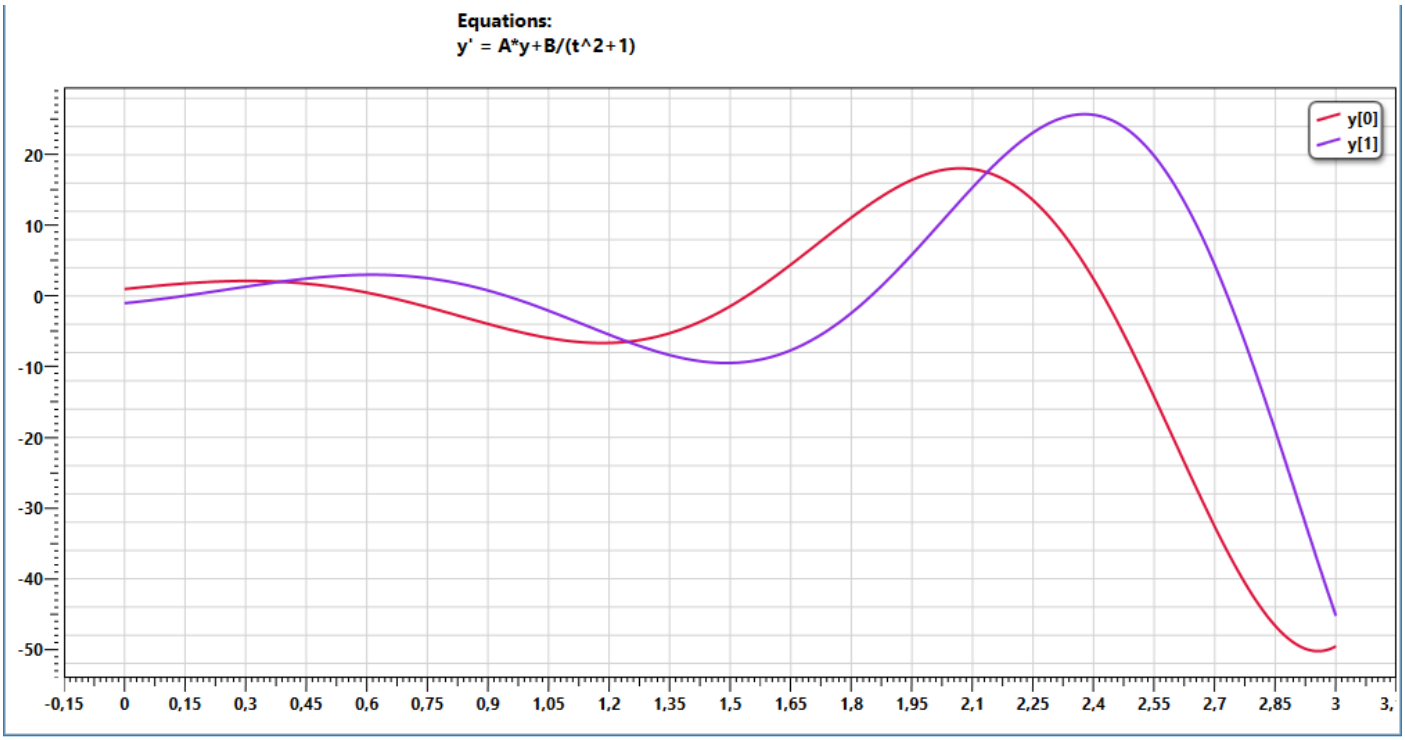
M'*A/(M'*M) = (0.459953357586789 0.163947437790482 0.113927221302585)

The code above demonstrates that symbolic vector and matrix operations allow solving complicated data processing problems with small source code. The formulae for the solution are compact and straightforward. Such problems cannot be solved using ‘scalar’ formula parsers and calculators (that do not support vector and matrix operations).
And the last example concerns numerical methods for solving systems of **ordinary differential equations**. There are so called matrix differential equations (https://en.wikipedia.org/wiki/Matrix_differential_equation) – systems of ordinary differential equations, written in compact form of one equation using vector and matrix notations. The **ANALYTICS** framework includes tools for numerical methods, totally integrated with symbolic features: linear and nonlinear least squares approximation; 1D and 2D function integration; nonlinear equation systems solution; ordinary differential equation systems solution. Solution of initial value problem for a matrix ODE can be solved with the following code:

```
// Parameter variables:
double[,] ma = new double[2,2]
{
    {3, -4  },
    {4, -0.7}
};
Variable A = new RealMatrixVariable("A", ma); // A-matrix.
double[] vb = new double[2] { -1, 1 };
Variable B = new RealArrayVariable("B", vb); // B-vector.
Variable[] prms = new Variable[]{ A, B };

double[] y0 = new double[2] { 1.0, -1.0 }; // Initial conditions.

// Create the Vector ODE system with the specified parameters.
ODESystem ode = new VectorODE("t", "y", "A*y+B/(t^2+1)", 2, prms);

double t1 = 3.0; // The end value of the interval for solution.
int N = 1000;    // The number of steps for interval discretization
double[] t = null;
// Use the Runge-Kutta 4 solver.
ODESolver solver = new RungeKutta4Solver();
```

```
// Solve the problem with the specified initial condition,
// integration interval and number of discretization steps.
double[][] y = solver.Solve(ode, y0, t1, N, ref t);
```

The result of the solution shown on the picture:



The source code of the examples (VS 2010 solution) can be found in the download for the article.

Conclusions

This article introduced the capabilities of the symbolic framework **ANALYTICS**. The framework is written entirely in C# and allows easily introduce new operations for vector and matrix data and integrate the framework with any application. Vector and matrix operations are the important part of many data processing applications, and symbolic vector and matrix operations allow to solve complicated data processing problems with small source code. The **ANALYTICS** framework includes many numerical algorithms, totally integrated with the symbolic capabilities: linear and nonlinear least squares approximation; 1D and 2D function integration; nonlinear equation systems solution; ordinary differential equation systems solution. The symbolic framework **ANALYTICS** can be found here http://sergey-l-gladkiy.narod.ru/index/analytics/0-13.