Symbolic evaluations and numerical methods

Sergey L. Gladkiy

Introduction

Numerical methods are one of the most important parts of modern software. They are used in mathematical modeling, physics, economics and other application. The applications which use numerical methods are commonly built with some numerical framework providing the computational realization of the methods. The numerical framework is integrated to the application's code and the application realizes a user interface for input data for the computational problems.

One of common numerical framework realization is that API functions require delegates for function evaluation. For an example, economical optimization problem requires a delegate for the objective function (profit). This approach is sufficient when the function is predefined and can be built directly in the application's code. But when the function can be defined by the application's user it is not simple task to provide the evaluation delegate to the numerical algorithm.

This article introduces an approach for creating numerical methods framework, integrated with symbolic evaluations. This approach provides easy integration of the framework to an application and allows to realize solution of user-defined numerical problems set up in the form of mathematical expressions.

Background

For explaining the advantages of symbolic framework and numerical methods integration, let us consider one common computational task – find definite integral of a function on an interval (https://en.wikipedia.org/wiki/Numerical_integration). We will consider the task from programming point of view – realization of a numerical method framework. The base class for numerical calculation of the integral could be the following:

```
/// <summary>
/// Base abstract class for 1D function integrators.
/// </summary>
public abstract class Integrator1D
{
    /// <summary>
    /// Integrates the function on the specified interval.
    /// </summary>
    /// <param name="f">Integrand function.</param>
    /// <param name="x1">Left variable value.</param>
    /// <param name="x2">Right variable value.</param>
    /// <param name="n">Number of integration nodes.</param>
    /// <returns>Integral value.</returns>
    public abstract double Integral(Function1D f, double x1, double x2, int n);
}
```

where the delegate is defined as

```
/// <summary>
/// 1D (univariate) function delegate.
/// </summary>
```

```
public delegate double Function1D(double x);
```

As can be seen from the code, integration process requires a delegate for the integrand function evaluation. This delegate can be provided as some predefined method in the application code only if the function is known on the code writing stage. But when the application must work with arbitrary functions, provided by the user's input, realization of the delegate is not easy, because the function cannot be written directly.

One of the approaches for solving the problem of user specified functions for numerical computation is using a symbolic framework. This framework must provide evaluation of arbitrary functions set up as math expression and should be incorporated with the numerical framework for the programming convenience.

From the application developer's point of view, the **advantages** of using symbolic evaluations with numerical methods are the following:

- Minimal code writing, because there is no need to write special classes for delegates (optimization objective function, integrand function and so on).
- Convenient data representation as math expressions, not as programming code.
- Easily using the user input data (string data) for manipulations in programming code.
- Easily transfer input data and the output results via network as strings, easily storing and serializing the data (no need to serialize delegates).
- Simple code for parametrized problems solution.

There is one **disadvantage** of using symbolic evaluations for numerical methods: the symbolic algorithms are supposed to be very slow, compared with the predefined functions, written directly with a programming language. So, the final computational effectiveness of the algorithm realization can be not enough for some critical cases of computational problems.

ANALYTICS framework

The **ANALYTICS** library is a general purpose symbolic framework for .NET. The library allows manipulations with symbolic mathematical expressions: evaluations, simplification and so on. Although the library includes advanced features, such as symbolic derivatives calculation, evaluation formula with arrays and matrices and many other, the common functionality is enough to integrate the framework with numerical methods.

Main library functionality encapsulated into the ***Translator*** class. Hereinafter it is supposed that the instance 'translator' of the ***Translator*** class has been created

```
Translator translator = new Translator();
```

Then we can add new variables to the instance or remove the existing ones:

```
string name = "x";
double v = 1.0;
translator.Add(name, v);
…
translator.Delete("x");
```

The common functionality for evaluating math expressions realized by the **Calculate** method of the class. The method can be easily used for simple, one time evaluations. For an example:

```
string f1 = "sin(a)^2+cos(a)^2";
double r = (double)translator.Calculate(f1);
string f2 = "2*exp(z)-I*sin(z/3)";
Complex c = (Complex)translator.Calculate(f2);
```

The ***Calculate*** method is rather slow because it parses string expression and creates internal structure to calculate result value. Thus, the usage of ***Calculate*** method is only recommended for single formula evaluation. For advanced case, when one formula must be evaluated several times for a set of variable values, we can use prebuilt formula object to avoid calling parsing algorithm for every evaluation. Here is the code example for evaluating a table of function values:

```
string s = "2*(sin(x)+cos(x))";
Formula f = translator.BuildFormula(s); // parsing string expression
Variable x = translator.Get("x");
double v = 0.0;
double[] ax = new double[101];
double[] ay = new double[101];
for (int i = 0; i <= 100; i++)
{
	ax[i] = v;
	x.Value = v; // setting new variable value
	ay[i] = (double)f.Calculate(); // calculating formula value for current x value
	v += 0.01;
}
```

In all above code examples it was supposed that the string expressions were syntactically correct. The symbolic framework must allow check the syntax of a formula, before using it. Here is common code sample of syntax checking with the **ANALYTICS** library:

```
string s = "2*(sin(x)+cos(x))";
try
{
	if (translator.CheckSyntax(s))
	{
		Formula f = translator.BuildFormula(s);
		if (f != null)
		{
			// here the formula calculation code
			// using f instance.
		}
	}
}
catch (Exception ex)
{
	// here the exception handling code.
}
```

The ***CheckSyntax*** method verifies that the string satisfies all syntactic rules: parenthesis parity, allowable sequences of symbols and others. If some rule is not satisfied – an exception thrown. The ***BuildFormula*** method verifies that all used in the formula elements exist (variables, functions and so on) and throws an exception if something not found. The described functionality is enough to integrate symbolic features with numerical framework.

Example of numerical algorithm with symbolic features

Now, let us consider an example of using symbolic framework to provide evaluation function for the numerical method. We are still using the ***Integrator1D*** class, described above. We need to provide a ***Function1D*** delegate for its calculation algorithm. This is realized as special class, which gets a math expression of the function as a string and provides the evaluation delegate for it. We start with the base abstract class for a multivariate function (function of several variables), and then inherit the class for univariate delegate. Here is the code of the base abstract class:

```
/// <summary>
/// Base abstract class for Symbolic function.
/// </summary>
public abstract class SymbolicFunction
{
    protected string[] _variables;
    protected Translator _translator;
    protected string _value;
    protected Formula _formula;

    protected virtual void CreateVariables(string[] v)
    {
        _variables = v;
        int n = v.Length;
        for (int i = 0; i < n; i++)
        {
            _translator.Add(v[i], 0.0);
        }
    }

    protected virtual void BuildFormula()
    {
        _formula = _translator.BuildFormula(_value);
    }

    /// <summary>
    /// Symbolic function expression.
    /// </summary>
    public string Value
    {
        get { return _value; }
    }

    /// <summary>
    /// Evaluator.
    /// </summary>
    public Translator Evaluator
    {
        get { return _translator; }
    }

    public SymbolicFunction(string[] v, string f)
```

```
    {
        _translator = new Translator();
        CreateVariables(v);
        _value = f;
        BuildFormula();
    }
}
```

The class's constructor receives the names of variables for the function ('**v**' array) and the expression for the function ('**f**' string). It creates an instance of the ***Translator*** classe and adds variables with specified names ('**CreateVariables'** method). Then it builds the formula object for specified expression.
The inherited class for univariate function is the following:

```
/// <summary>
/// Symbolic 1D (1 variable) function.
/// </summary>
public class SymbolicFunction1D: SymbolicFunction
{
    /// <summary>
    /// Constructor for Symbolic 1D function.
    /// </summary>
    /// <param name="v">Variable name.</param>
    /// <param name="f">Symbolic function expression.</param>
    public SymbolicFunction1D(string v, string f) : base(new string[1]{v}, f)
    {
    }

    /// <summary>
    /// Evaluates the function for the specified variable value.
    /// </summary>
    public double F(double v)
    {
        _translator.Variables[0].Value = v;
        double result = (double)_formula.Calculate();
        return result;
    }
}
```

It overrides the constructor, because an univariate function depends on one variable only (constructor receives one variable name '**v**'). The class also adds new method '**F**' – this is the delegate of the evaluation function, required for numerical algorithms.
It should be noted here, that even we realized the special class for the delegate, it differs from the case, when we write the delegate directly for the computational problem. In the first case we have written one class for all possible functions, in the last case we need to write new delegate for every computational problem. Here is an example code of creating the symbolic function and evaluating it for one argument value:

```
SymbolicFunction1D f = new SymbolicFunction1D("x", "3*sin(x)*cos(2*x");
double y = f.F(0.5);
```

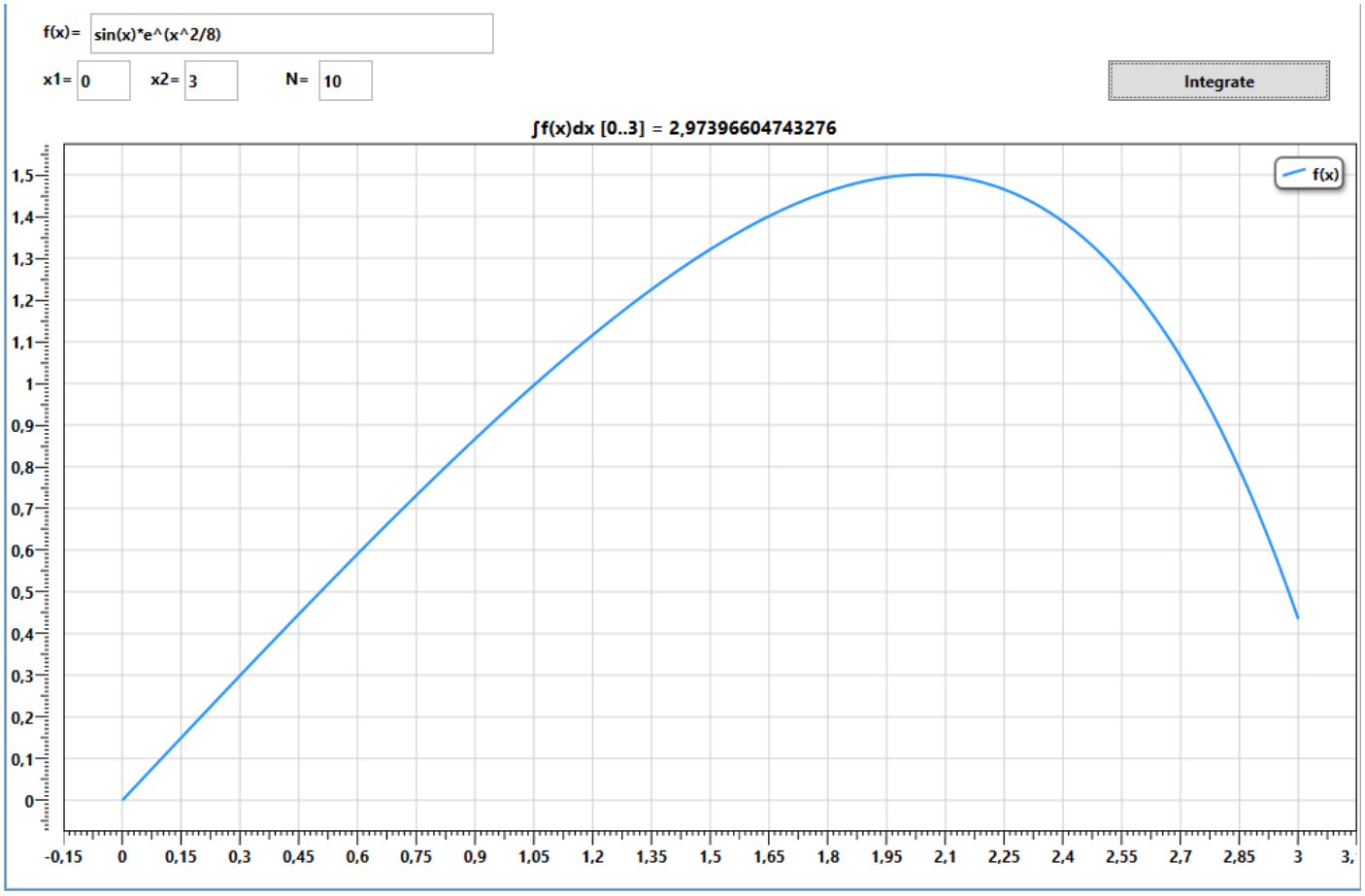
Thus, the evaluation function can be created for any math expression and then used in numerical calculations.

Now we are ready to create an application to calculate definite integrals for arbitrary function, provided by user input (as string for math expression). As the realization of numerical algorithms is out of the article's scope, we suppose here that we have some realization of the ***Integrator1D*** class. Then, the code for the application could be the following (total code of the WPF application can be found in the download):

```
string f = textIF.Text; // Integrable function provided with the user input.
double x1 = double.Parse(textIx1.Text); // Left value of the integration interval.
double x2 = double.Parse(textIx2.Text); // Right value of the integration interval.
int n = int.Parse(textIN.Text); // Number of integration nodes.

Integrator1D integrator = new Gauss2NodeIntegrator(); // Create the integrator instance.
SymbolicFunction1D sf1D = new SymbolicFunction1D("x", f); // Create the symbolic function instance.
double ivalue = integrator.Integral(sf1D.F, x1, x2, n);   // Integrate the function.
```

Here is the result of a function integral calculation with visualization:

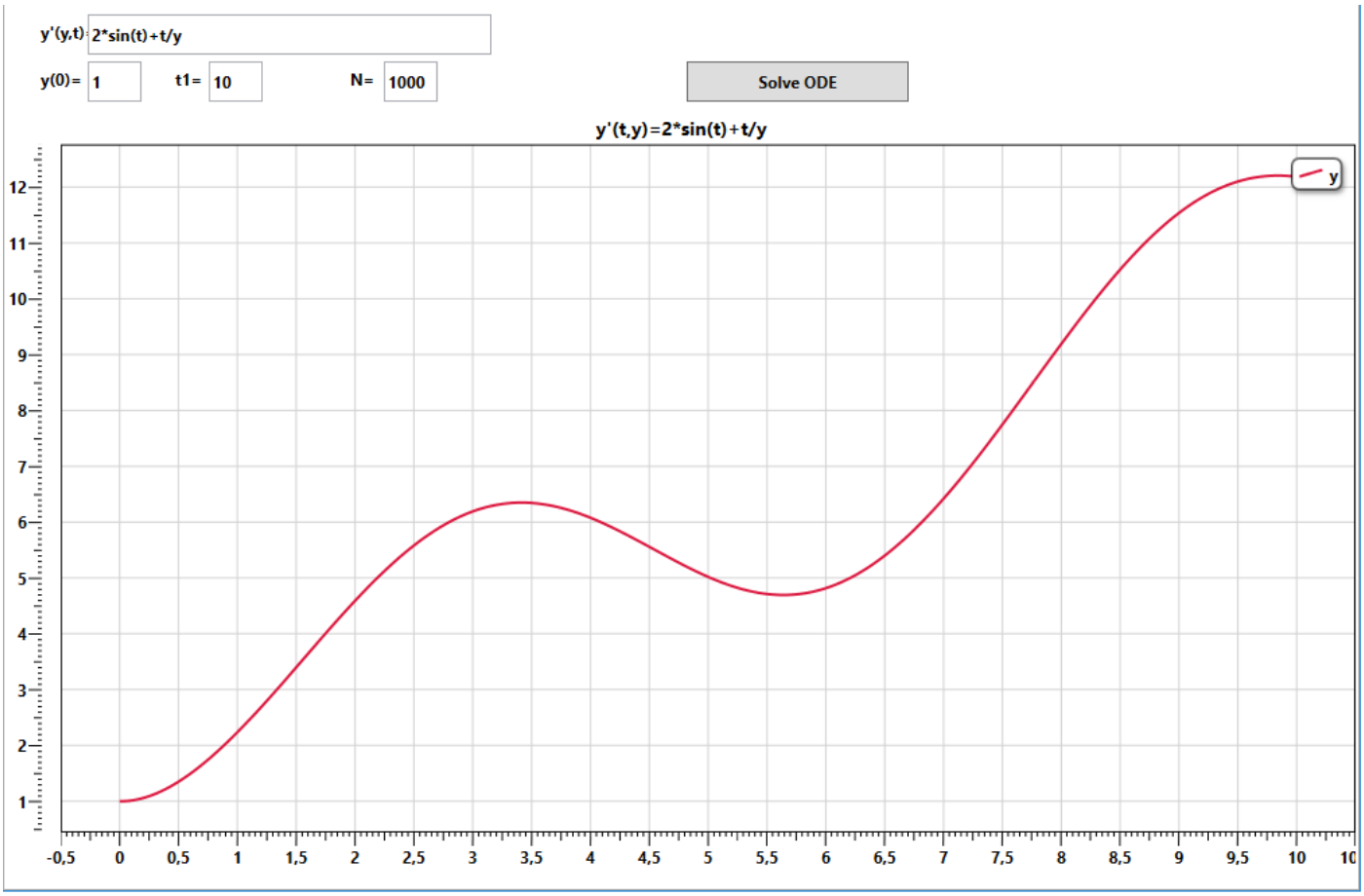


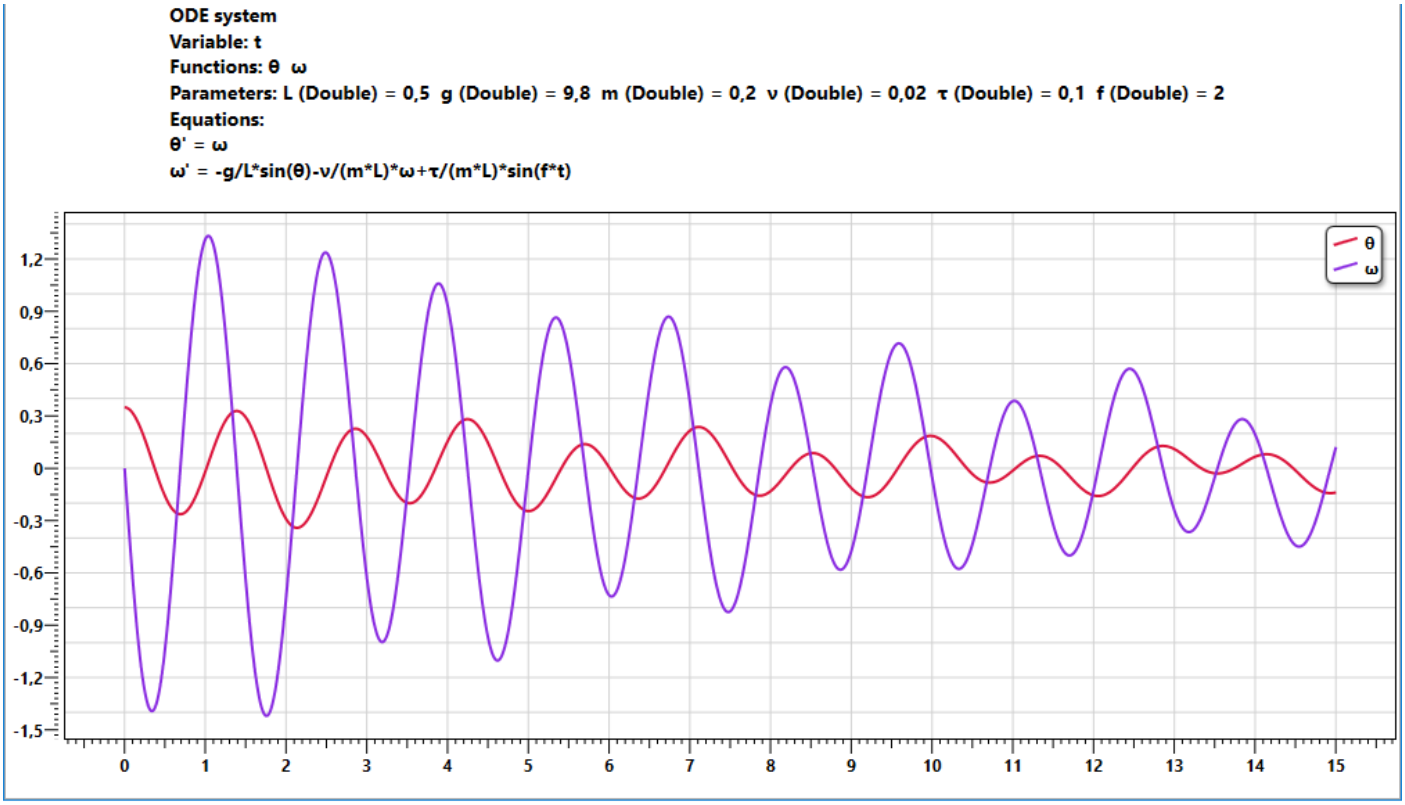
This examples demonstrates that using symbolic capabilities with numerical methods allows easily creating applications for solving user defined computational problems.

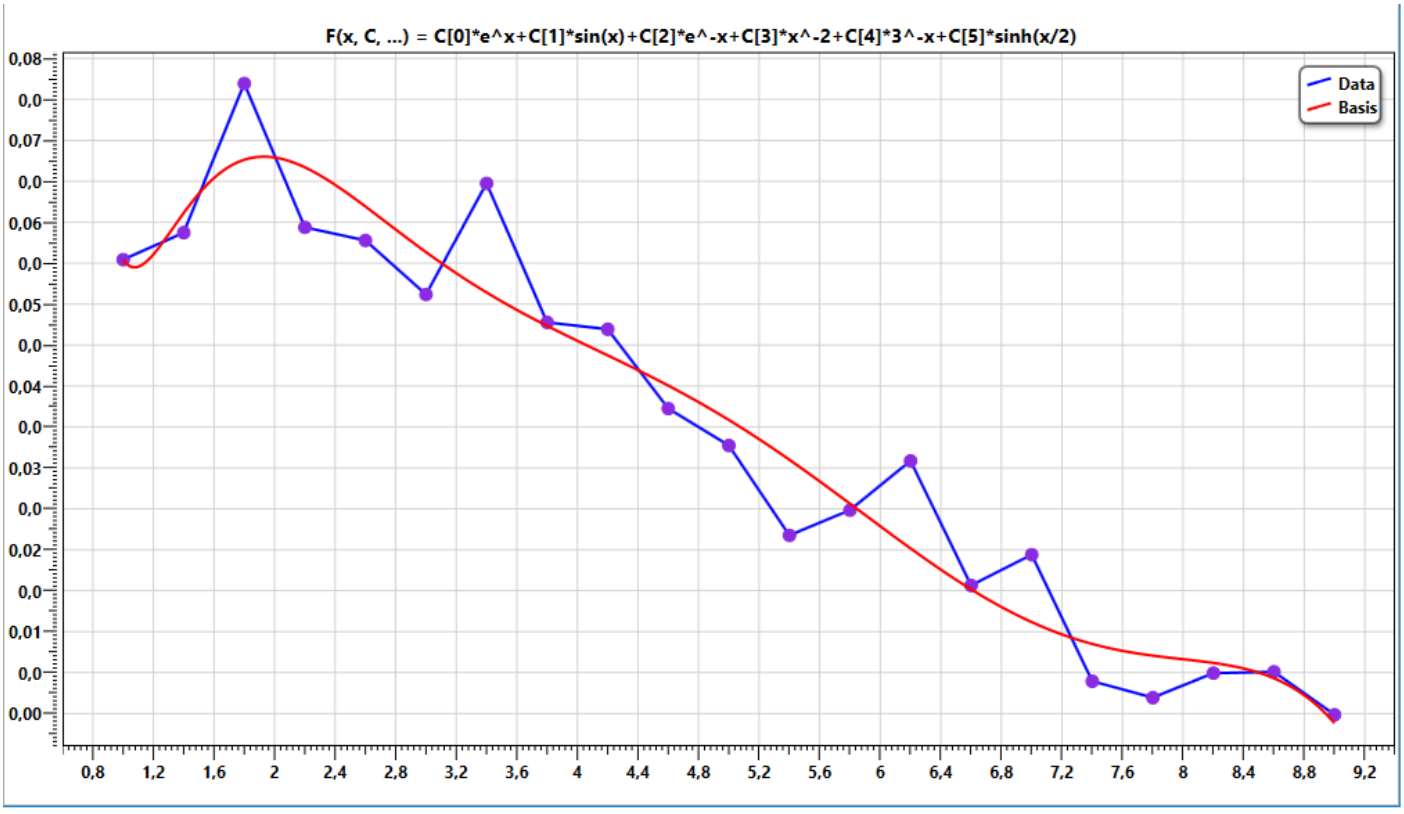
It should be also noted that our ***SymbolicFunction1D*** class can be used not only for numerical integration algorithm but for other numerical methods: finding roots of nonlinear equations, finding extremums of functions and so on.

We can also inherit the 2D function from the base class ***SymbolicFunction*** and then use it in numerical applications for functions of two variables.

Analogous approach of using symbolic evaluations together with numerical algorithm can be applied for other applications: solving ordinary differential equations (initial value problems), least squares approximation, solving systems of nonlinear equations, solving optimization problems and many others. As the examples of the approach realization, here are the results of solving the following problems: ordinary differential equation, system of two ODEs, linear least squares approximation (all equations and functions for the problems were defined as math expressions).







The source code of the examples (VS2010 solution) can be found in the download for the article.

Conclusions

The article introduced an approach of using symbolic evaluations with numerical algorithms for creating applications which deals with various computational problems. The approach is useful when the application must solve problems for user defined equations. There are some advantages of using symbolic evaluations: minimal code writing; convenient data representation as math expressions; easy using the user input data (string data); easy store and transfer input data and output results as strings; easy integration in various applications. The symbolic framework **ANALYTICS**, used for realizing analytical evaluations for numerical algorithms, can be found here http://sergey-l-gladkiy.narod.ru/index/analytics/0-13. There are some other examples of integration the symbolic algorithms with numerical frameworks: **NMath** (http://sergey-l-gladkiy.narod.ru/index/nmath-analytics/0-21), **ILNumerics** (http://sergey-l-gladkiy.narod.ru/index/il_analytics/0-18), **Extreme Optimization** (http://sergey-l-gladkiy.narod.ru/index/extreme-analytics/0-22).