Exploring parametric ODE with symbolic C# framework

Sergey L. Gladkiy

Introduction

Ordinary differential equations (**ODE**) used in many applications for physical, economical and other problem solution. The equations commonly describe the time evolution of some system from known initial state – **initial value problem** (**IVP**) (https://en.wikipedia.org/wiki/Initial_value_problem). Some initial value problems for simplest differential equations can be solved analytically - the result is some math expression for the unknown function. Then the solution can be totally analyzed for its properties. But in the most real cases the solution of an initial value problem can be found only with numerical methods, and then the solution is an array of numbers for the function value. There are many numerical algorithms for solving initial value problems – Runge-Kutta (https://en.wikipedia.org/wiki/Runge%E2%80%93Kutta_methods), Dormand-Prince (https://en.wikipedia.org/wiki/Dormand%E2%80%93Prince_method) and so on.

This article explains how parametric ODE's can be explored with the symbolic framework **ANALYTICS**. The framework allows solving initial value problems for ordinary differential equations with parameters, and so one can explore how the parameter value influences the system behavior.

Background

The **ANALYTICS** library is a symbolic framework written in C#. The main goal of the library is providing simplest way for evaluating math formula. The library also allows manipulations with symbolic mathematical expressions, such as simplification and symbolic derivative calculation.

Main library functionality encapsulated into the ***Translator*** class. The class allows to add new variables and then evaluate math expressions with them. First we must create an instance of the class:

```
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");
```

We can easily evaluate math expressions using the **Calculate** method of the class. 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);
```

Although the library is a **symbolic** framework, it also contains many numerical tools, such as: linear and nonlinear least squares approximation; 1D and 2D function integration; nonlinear equation systems solution; ordinary differential equation systems solution. The advantage of the framework is that all numerical tools are totally integrated with symbolic features. This allows exploring various parametric problems with the numerical tools. More information about the **ANALYTICS** symbolic framework can be found in the library's guide (http://sergey-l-gladkiy.narod.ru/ana_docs/ANALYTICS_C_manual.en.pdf).

Realization

Now we are going to realize some classes for describing the process of initial value problem solution. Here should be noted that all existing numerical algorithms intended for solving systems of ODEs of the **first order**. The systems of equations of higher orders must be converted to larger systems of equations of the first order (https://en.wikipedia.org/wiki/Ordinary_differential_equation#Reduction_to_a_first-order_system).
First we must define base abstract class of the ODE system, which can be the following:

```
/// <summary>
/// Base abstract class for Ordinary Differential Equation system (1st order).
/// </summary>
public abstract class ODESystem
{
    protected abstract int GetDimension();

    /// <summary>
    /// Dimension (number of equations).
    /// </summary>
    public int Dimension
    {
        get { return GetDimension(); }
    }

    /// <summary>
    /// Evaluates the system's equations.
    /// y1' = f1(t, y1, y2, ...)
    ///  ...
    /// yn' = fn(t, y1, y2, ...)
    /// </summary>
    public abstract double[] Evaluate(double t, double[] y);
}
```

The class encapsulates one property ***Dimension*** (number of equations in the system) and one method ***Evaluate*** which receives the time value '**t**' and the function values '**y**' and returns the array of values for the equations (right parts).
The next step is defining the base class for ODE solver. Here is the code of the class:

```
/// <summary>
/// Base abstract class for Initial Value Problem solver (for ODE system of the 1st order).
/// </summary>
public abstract class ODESolver
{
    /// <summary>
    /// Solves the Initial Value problem for the ODE system on the specified time interval.
    /// </summary>
```

```
    /// <param name="system">The ODE system</param>
    /// <param name="y0">The initial function values (at t=0)</param>
    /// <param name="t1">The end interval value.</param>
    /// <param name="N">The number of discretization steps.</param>
    /// <param name="t">The discrete variable values.</param>
    /// <returns>Function values found for the discrete variable values.</returns>
    public abstract double[][] Solve(ODESystem system, double[] y0, double t1, int N, ref double[]
t);
}
```

The only abstract method of the class receives as parameters the ODE system '**system**', initial values of the unknown functions '**y0**', the end of the solution time interval '**t1**' (initial time value is always 0), the number of time steps for numerical solution. The method returns calculated function values for every time step, and as the **ref** parameter '**t**' – time values for every time step.

The abstract solver class hierarchy can be divided into several branches, depending on the solver types. The most used solver types are so called **explicit (or direct) solvers**. They use information of one time step for evaluating the next time step values (https://en.wikipedia.org/wiki/Explicit_and_implicit_methods). The abstract class for the direct solver is the following:

```
/// <summary>
/// Base abstract class for Direct solver using the information on the one step
/// to calculate the function values on the next step
/// </summary>
public abstract class DirectODESolver: ODESolver
{
    /// <summary>
    /// Makes one solution step.
    /// </summary>
    /// <param name="system">The ODE system</param>
    /// <param name="yi">The current function values (at t=ti)</param>
    /// <param name="ti">Current variable value.</param>
    /// <param name="dt">The step value.</param>
    /// <returns>Function values found for the t=ti+dt.</returns>
    public abstract double[] Step(ODESystem system, double[] yi, double ti, ref double dt);
}
```

Direct solvers can be divided into '**fixed-step**' and '**automatic-step**' ones. The fixed step solver makes all time steps with one, fixed time step value. Here is the code for the class:

```
/// <summary>
/// Base abstract class for Fixed Step Direct solver (dt = const).
/// </summary>
public abstract class FixedStepSolver : DirectODESolver
{
    /// <summary>
    /// Solves the problem with the direct algorithm (with dt = const).
    /// </summary>
    /// <param name="system">The ODE system</param>
    /// <param name="y0">The initial function values (at t=0)</param>
    /// <param name="t1">The end interval value.</param>
    /// <param name="N">The number of discretization steps.</param>
    /// <param name="t">The discrete variable values.</param>
    /// <returns>Function values found for the discrete variable values.</returns>
```

```
    public override double[][] Solve(ODESystem system, double[] y0, double t1, int N, ref double[]
t)
    {
        double[][] result = new double[N + 1][];
        double dt = t1 / N;
        t = new double[N + 1];
        t[0] = 0.0;
        result[0] = y0;
        for (int i = 1; i <= N; i++)
        {
            t[i] = dt * i;
            result[i] = Step(system, result[i - 1], t[i - 1], ref dt);
        }

        return result;
    }
}
```

Finally, we can realize the solver for some concrete numerical algorithm. Maybe the most known algorithm for solving ODE is the Runge-Kutta 4-th order (https://en.wikipedia.org/wiki/Runge%E2%80%93Kutta_methods). Its realization is the following:

```
/// <summary>
/// The Runge-Kutta 4-th order solver.
/// </summary>
public class RungeKutta4Solver: FixedStepSolver
{
    /// <summary>
    /// Makes one solution step with the Runge-Kutta 4-th order formula.
    /// </summary>
    public override double[] Step(ODESystem system, double[] yi, double ti, ref double dt)
    {
        const double c = 1.0 / 6.0;
        Vector yn = new Vector(yi);

        Vector k1 = dt*(new Vector(system.Evaluate(ti, yi)));
        Vector k2 = dt*(new Vector(system.Evaluate(ti + 0.5*dt, (yn + 0.5*k1).Data)));
        Vector k3 = dt*(new Vector(system.Evaluate(ti + 0.5*dt, (yn + 0.5*k2).Data)));
        Vector k4 = dt*(new Vector(system.Evaluate(ti + dt, (yn + k3).Data)));

        Vector yn1 = yn + c*(k1 + 2.0*(k2 + k3) + k4);

        return yn1.Data;
    }
}
```

Here is some ‘magic’ in the code – using the ***Vector*** class. This class is a wrapper for array data **double[]** with overloaded algebraic operations. This allows write ‘vector’ Runge-Kutta’s formula for the system of equations in the form of simple ‘scalar’ formula for one equation.

One thing left is to integrate the realized numerical solvers with the symbolic capabilities of the **ANALYTICS** framework. For making this we must realize special class of ‘analytical’ ODE system. This class must be constructed with the equations, defined as symbolic math expressions (string data), and, using the ***Translator*** class, override the ***Evaluate*** method of the class. Also, the class must introduce the capability of introducing parameters for the equations.

The realization of ***AnalyticalODE*** class includes some additional class hierarchy and is not written here for simplifying reading. Instead, here is the interface code of the final class ***ScalarODE***:

```
/// <summary>
/// Scalar Analytical ODE system (defined by the set of scalar equations).
/// </summary>
public class ScalarODE: AnalyticalODE
{
    /// <summary>
    /// Costructor for ODE system from the set of scalar equations.
    /// y1' = f1(t, y1, y2, ...)
    ///  ...
    /// yn' = fn(t, y1, y2, ...)
    /// </summary>
    /// <param name="v">Variable name, commonly 't'.</param>
    /// <param name="fv">Unknown Function names, commonly 'y1', 'y2', ...</param>
    /// <param name="equations">Equations (f1, f2, ..., fn).</param>
    /// <param name="parameters">Parameter variables.</param>
    public ScalarODE(string v, string[] fv, string[] equations, Variable[] parameters) : base(v, fv,
equations, parameters)
    {
    }
}
```

As can be seen from the code, the instance of the analytical scalar ODE system is constructed using the analytical math expressions '**equations**' and allows introducing parameter variables for the equations. The parameter values can be changed via the interface of analytical ODE class. So, one instance of the ODE system can be used for solving it with different parameter values and exploring the system's behavior.

Example of exploring parametric ODE

Now, let us consider an example of solving a parametric ODE equation. We will explore the following simple parametric ODE:

$$\frac{\mathrm{d}}{\mathrm{d}\,t}y = A\cdot\sin(a\cdot t) + \frac{B\cdot t}{y^{b}}$$

with initial condition y(0)=1 on the time interval t=[0, 10]. The problem is parametric because there are parameters '**A**', '**a**', '**B**' and '**b**' in the equation. Our goal is exploring how the solution (function **y(t)**) depends on values of the parameters.

First we need to create the instance of analytical ODE object for the equation. The code is the following:

```
// Parameter variables:
Variable A = new RealVariable("A", 1.0); // A-parameter.
Variable a = new RealVariable("a", 1.0); // a-parameter.
Variable B = new RealVariable("B", 0.1); // B-parameter.
Variable b = new RealVariable("b", 1.0); // b-parameter.
Variable[] prms = new Variable[] { A, a, B, b }; // parameters
string[] vnames = new string[] { "y" };    // function names
string[] equations = new string[] { "A*sin(a*t)+B*t/y^b" }; // equations

// Create the ODE system with the specified equations and parameters.
```

```
AnalyticalODE ode = new ScalarODE("t", vnames, equations, prms);
```

The code introduces variables for all parameters with some initial values, specifies the set of equations (in this case it is one equation only) and then creates the instance of the analytical ODE with the parameters. For exploring the equation we also need a solver instance, initial conditions and time interval:

```
// Use the Runge-Kutta 4-th order colver.
ODESolver solver = new RungeKutta4Solver();
// Initial conditions.
double[] y0 = new double[1] { 1.0 };
double t1 = 10.0; // The end value of the interval for solution.
int N = 1000;    // The number of steps for interval discretization
double[] t = null;
```

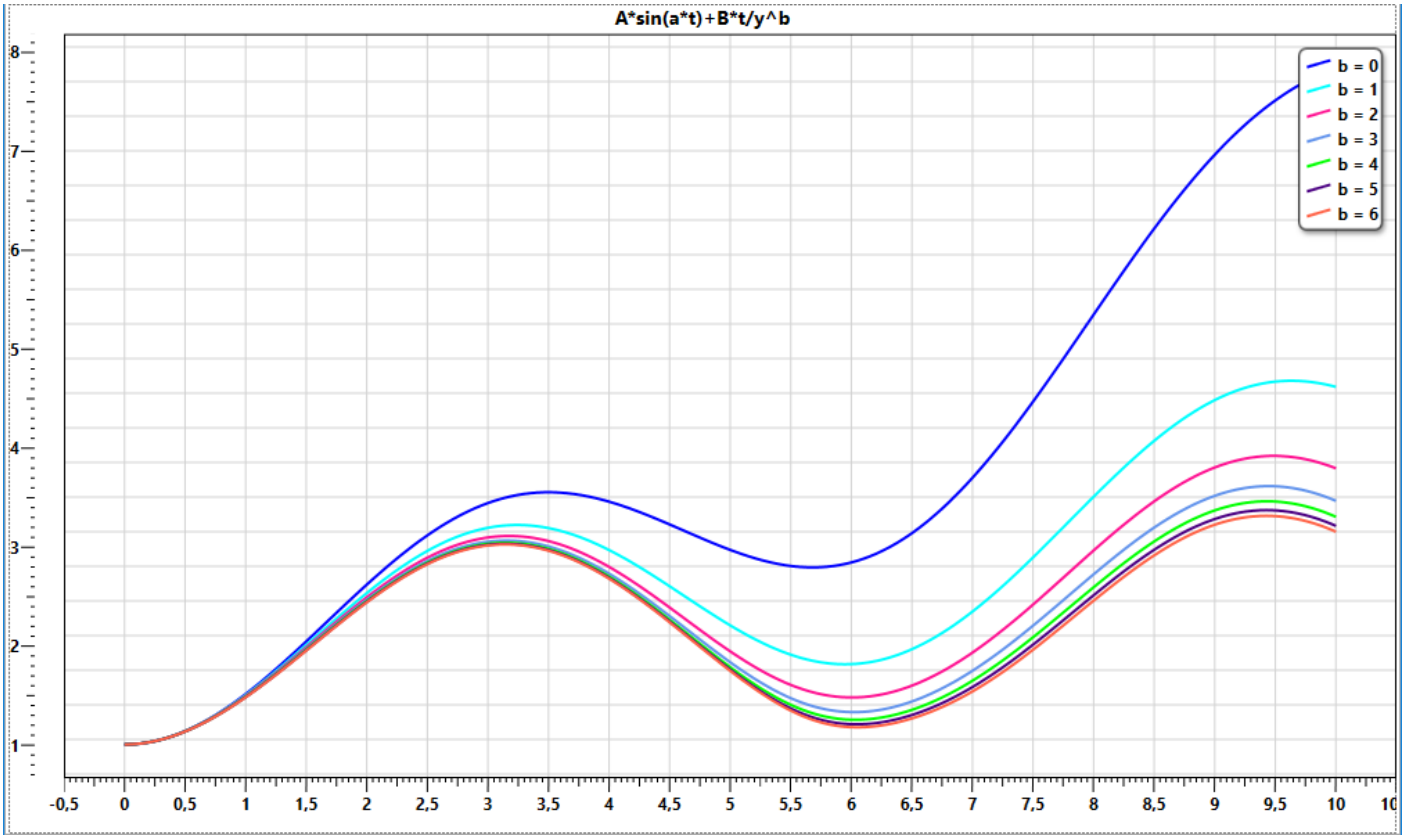
Now we are ready to solve the equation. First, we are going to explore the system's behavior depending on the '**b**' parameter value. Here is the code for the solution:

```
// b-parameter exploring...
for (int i = 0; i < 7; i++)
{
    // Set the exploring parameter value.
    double bv = (double)i;
    ode.Parameters("b").Value = bv;

    // Solve the problem with specified 'b' value.
    double[][] y = solver.Solve(ode, y0, t1, N, ref t);

    // Use the solution 'y'…
}
```

The code above changes value of the '**b**' parameter in the loop, and solves the system for the specified value. The result of the solution for various '**b**' values shown on the picture below:



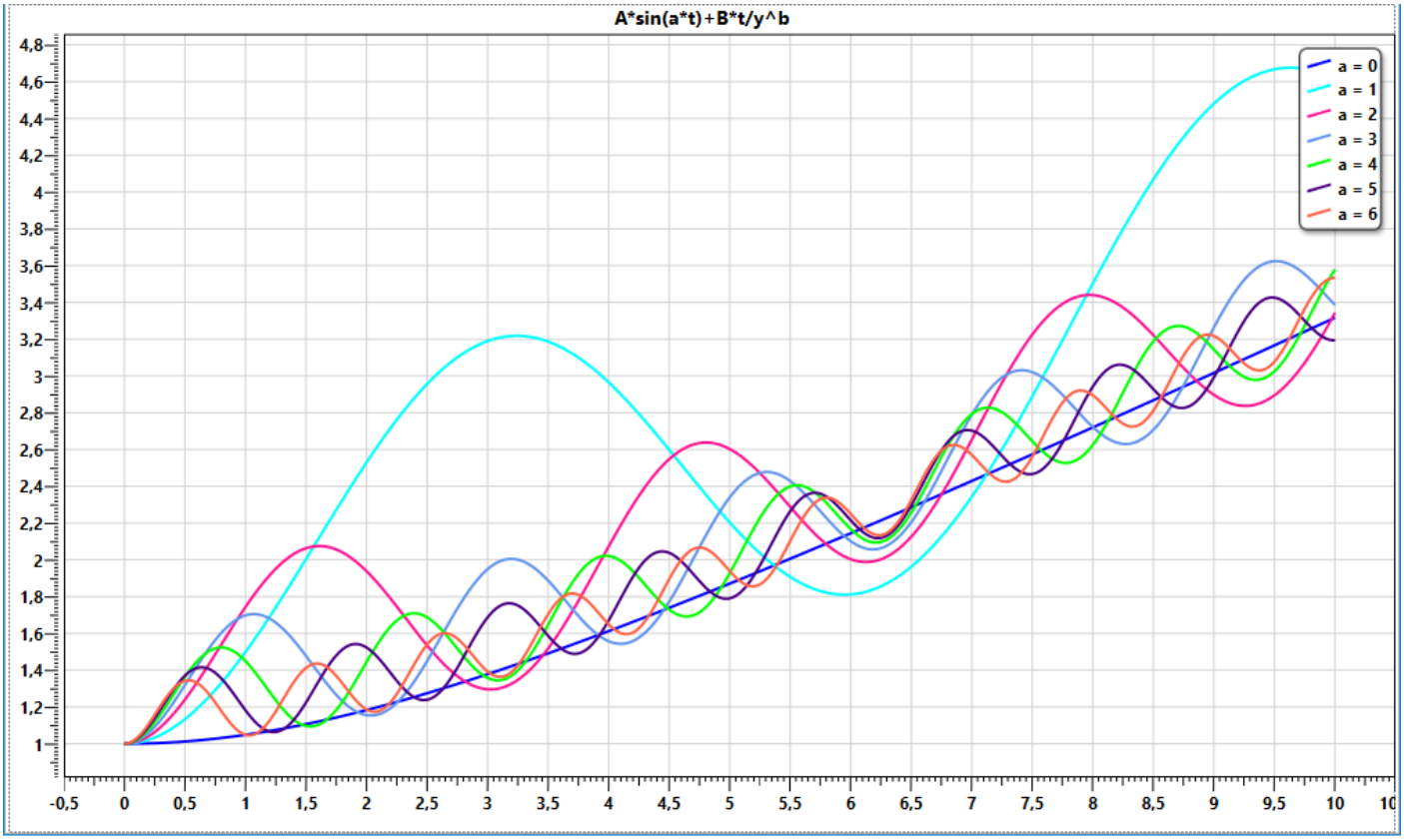
The same code can be used to explore the system's behavior, depending on the '**a**' parameter:

```
// a-parameter exploring...
for (int i = 0; i < 7; i++)
{
    // Set the exploring parameter value.
    double av = (double)i;
    ode.Parameters("a").Value = av;

    // Solve the problem with specified 'a' value.
    double[][] y = solver.Solve(ode, y0, t1, N, ref t);

    // Use the solution 'y'…
}
```

And here is the result:



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

Conclusions

This article introduced an approach of realizing numerical solvers for initial value problems with using symbolic (analytical) evaluations. The symbolic capabilities of the **ANALYTICS** framework allow creating parametric ODE systems and then explore their behavior depending on the parameter's value. Other advantages of using the numerical ODE tool with the symbolic calculus are: minimal code writing (there is no need to write special classes for evaluation delegates); convenient data representation for the developer and user as math expressions, not as programming code; easily using the user input data (string data) for manipulations in programming code; easily transfer input data via network as strings, easily storing and serializing the data. The symbolic framework **ANALYTICS** can be found here http://sergey-l-gladkiy.narod.ru/index/analytics/0-13.