Solving nonlinear equation systems with analytical derivative calculations in C#

Sergey L. Gladkiy

Introduction

There are many libraries for solving nonlinear equation systems. They realize well-known mathematical algorithms – Newton-Raphson method, Levenberg-Marquardt method, Powell's Dog Leg method and so on. The algorithms require calculation not only values of system equations at some point, but also the values of gradient, Jacobian or Hessian of the system, that are all some type of derivatives of the system. Common realizations of computational libraries uses numerical calculation of the derivatives – finite difference. This article introduces alternative approach, when the nonlinear system is set up in the form of analytical expressions and derivative expressions also provided in analytical form. Calculations of analytical expressions and derivation process carried out with ANALYTICS C# library.

Background

For understanding analytical realization of nonlinear equations system solution, let us consider the general programming environment for the problem. In general case there must be 2 main classes: *NonlinearSystem* – represents nonlinear equations system and *NonlinearSolver* – represents nonlinear solver. The presented realization of the classes implemented in the MATHEMATICS library as the following:

```
/// <summary>
/// Base class for Nonlinear Equation system.
/// </summary>
public abstract class NonlinearSystem
{
    /// <summary>
    /// Dimension (the number of equations).
    /// </summary>
    public int Dimension
    {
        get;
    }

    /// <summary>
    /// The system supports derivatives.
    /// </summary>
    public bool DerivativeSupported
    {
        get;
    }

    /// <summary>
    /// Calculates equation values for given variable values.
    /// </summary>
    /// <param name="x"></param>
    /// <param name="y"></param>
    /// <returns></returns>
    public abstract bool Calculate(double[] x, ref double[] y);

    /// <summary>
    /// Calculates the system's Jacobian.
    /// (must be overridden in inherited classes if derivative calculation supported)
```

```
        /// </summary>
        /// <param name="x"></param>
        /// <param name="J"></param>
        /// <returns></returns>
        public virtual bool Jacobian(double[] x, ref double[][] J)
        {…}
    }

    /// <summary>
    /// Abstract nonlinear solver.
    /// </summary>
    public abstract class NonlinearSolver
    {
        /// <summary>
        /// Solution method.
        /// </summary>
        /// <param name="system"></param>
        /// <param name="x0"></param>
        /// <param name="opt"></param>
        /// <param name="x"></param>
        /// <returns></returns>
        public abstract SolutionResult Solve(NonlinearSystem system, double[] x0,
                                             SolverOptions opt, ref double[] x);
    }
```

As for common nonlinear solvers are iterative ones, the inherited class *IterativeSolver* provided and it overrides Solve method in the form of iteration step sequence:

```
    /// <summary>
    /// Base Abstract Iterative Solver.
    /// </summary>
    public abstract class IterativeSolver : NonlinearSolver
    {
        /// <summary>
        /// Iteration.
        /// </summary>
        /// <param name="system"></param>
        /// <param name="x0">Current x values</param>
        /// <param name="x1">Next (calculated) x values</param>
        /// <returns></returns>
        protected abstract bool Iteration(NonlinearSystem system, double[] x0, ref double[] x1);

        /// <summary>
        /// Common Iterative Solution method.
        /// </summary>
        /// <param name="system"></param>
        /// <param name="x0"></param>
        /// <param name="opt"></param>
        /// <param name="x"></param>
        /// <returns></returns>
        public override SolutionResult Solve(NonlinearSystem system, double[] x0,
                                             SolverOptions opt, ref double[] x)
        {
            … iteration process here
        }
    }
```

The MATHEMATICS library realizes several final solver classes. The *NewtonRaphsonSolver* class used in all examples as the solver, requiring the derivative (Jacobian) calculations.

Realization

For realization of the approach with analytical derivative calculation, an analytical evaluation engine is required. The ANALYTICS C# library used for that purpose. This library provides not only analytical expression evaluation, but also analytical derivative calculation. Using this library the following descendant class for *NonlinearSystem* realized:

```
/// <summary>
/// System of Nonlinear Analytical Equations.
/// </summary>
public class AnalyticalSystem: ConstructedSystem
{
    #region Analytical data members
    protected Translator translator;
    protected string[] functions;
    protected string[,] dfunctions;
    protected Formula[] formulae;
    protected Formula[,] fderivatives;
    #endregion Analytical data members

    /// <summary>
    /// Creates translator and adds Real variables to it.
    /// </summary>
    /// <param name="variables"></param>
    /// <returns></returns>
    protected bool AssignVariables(string[] variables)
    {
        if (variables == null || variables.Length == 0) return false;

        translator = new Translator();
        int l = variables.Length;
        for (int i = 0; i < l; i++)
        {
            if (!translator.Add(variables[i], (double)0.0))
            {
                throw new InvalidNameException(variables[i]);
            }
        }

        return true;
    }

    /// <summary>
    /// Creates formulae for equations and their derivatives.
    /// </summary>
    /// <param name="f"></param>
    /// <returns></returns>
    protected bool AssignFunctions(string[] f)
    {
        if (f == null || f.Length == 0) return false;

        int l = f.Length;
        functions = new string[l];
        formulae = new Formula[l];
        for (int i = 0; i < l; i++)
```

```
    {
        if (!translator.CheckSyntax(f[i]))
        {
            functions = null;
            formulae = null;
            return false;
        }

        functions[i] = f[i];
        formulae[i] = translator.BuildFormula(functions[i]);
        if (formulae[i].ResultType != typeof(double))
        {
            Type t = formulae[i].ResultType;
            functions = null;
            formulae = null;
            throw new WrongArgumentException("Function must return real value.",
                                             typeof(double), t);
        }
    }

    dfunctions = new string[l, l];
    fderivatives = new Formula[l,l];
    try
    {
        for (int i = 0; i < l; i++)
        {
            for (int j = 0; j < l; j++)
            {
                dfunctions[i, j] = translator.Derivative(functions[i],
                                                         translator.Variables[j].Name);
                fderivatives[i, j] = translator.BuildFormula(dfunctions[i, j]);
            }
        }
    }
    catch (Exception)
    {
        // if some derivative could not be calculated -
        // the system does not support Jacobian.
        fderivatives = null;
    }

    return true;
}

/// <summary>
/// Assigns variable values.
/// </summary>
/// <param name="values"></param>
protected void AssignVariableValues(double[] values)
{
    int l = values.Length;
    for (int i = 0; i < l; i++)
    {
        translator.Variables[i].Value = values[i];
    }
}

/// <summary>
/// Calculates equation result for current variable values.
```

```
/// </summary>
/// <param name="i">Equation number</param>
/// <returns></returns>
protected double Equation(int i)
{
    return (double)formulae[i].Calculate();
}

/// <summary>
/// Calculates derivative result for current variable values.
/// </summary>
/// <param name="i">Equation number</param>
/// <param name="j">Variable number</param>
/// <returns></returns>
protected double Derivative(int i, int j)
{
    return (double)fderivatives[i,j].Calculate();
}

/// <summary>
/// Creates equation (and derivative) delegates
/// based on the created formulae objects.
/// </summary>
protected void CreateEquations()
{
    int l = formulae.Length;
    equations = new Equation[l];
    for (int i = 0; i < l; i++)
    {
        // DO NOT remove temp variable,
        // using 'i' directly leads to incorrect lambda.
        int temp = i;

        equations[i] = (double[] x) =>
            {
                AssignVariableValues(x);
                return Equation(temp);
            };
    }

    // if formulae of derivatives are not assigned -
    // system will not support Jacobian.
    if (fderivatives != null)
    {
        derivatives = new Derivative[l];
        for (int i = 0; i < l; i++)
        {
            // DO NOT remove temp variable,
            // using 'i' directly leads to incorrect lambda.
            int tempi = i;

            derivatives[i] = (int j, double[] x) =>
            {
                AssignVariableValues(x);
                return Derivative(tempi, j);
            };
        }
    }
}
```

```
        /// <summary>
        /// Constructor.
        /// </summary>
        /// <param name="variables">Variable names, the system must be resolved of.</param>
        /// <param name="f">System Equation Functions</param>
        public AnalyticalSystem(string[] variables, string[] f)
        {
            if (variables.Length != f.Length)
            {
                throw new WrongArgumentException("Variable count must be equal to equation count.",
                                                 variables.Length, f.Length);
            }

            if (!AssignVariables(variables))
            {
                return;
            }

            if (!AssignFunctions(f))
            {
                return;
            }

            CreateEquations();
        }
    }
```

Main features of the system realization can be seen from constructor and *AssignFunctions* method. System's constructor gets equations provided in analytical expressions form (array of strings). There is no need providing analytical expressions (or any lambda) for derivatives (Jacobian), because they are automatically calculated in *AssignFunctions* method by one line of code:

```
dfunctions[i, j] = translator.Derivative(functions[i], translator.Variables[j].Name);
```

There are some benefits of such approach. First, there is no need to provide lambda expressions not for system equations not for Jacobian. Second, no need to make derivation process for Jacobian functions, the process made automatically with analytical engine provided. Last, the calculations of analytical expressions gives more precise derivative value estimations than numerical finite difference, which lacks from the step choice.

Test examples

Two test examples provided in VS 2010 solution. The examples demonstrate using analytical system for solving geometry problems – finding intersection of curves in 2D space and surfaces in 2D space. The code of the 3D case:

```
// variables of the analytical system
string[] variables = { "x", "y", "z" };
// analytical expressions of the system's equations
string[] functions = { "x^2+y^2+z^2-1", // sphere
                       "x^2+y^2-z",     // paraboloid along the z axis
                       "-x+2*y^2+z^2"   // paraboloid along the x axis
                                     };
// creating nonlinear system instance - Analytical System
NonlinearSystem system = new AnalyticalSystem(variables, functions);
```

```
double[] x0;
SolverOptions options;
double[] result;
double[] expected;
SolutionResult actual;

// creating nonlinear solver instance - Newton-Raphson solver.
NonlinearSolver solver = new NewtonRaphsonSolver();

x0 = new double[] { 1.0, 1.0, 1.0 }; // initial guess for variable values
options = new SolverOptions()        // options for solving nonlinear problem
{
       MaxIterationCount = 100,
       SolutionPrecision = 1e-5
};

result = null;
// solving the system
actual = solver.Solve(system, x0, options, ref result);

expected = new double[] { 0.6835507455, 0.3883199288, 0.6180339887 }; // expected values
// printing solution result into console out
PrintNLResult(system, actual, result, expected, options.SolutionPrecision);
```

The result of the test execution is printed to the VS console output. For the previous code, the output is the following:

```
Analytical equations system
Dimension: 3
Support derivatives: True
Equations:
[0]=x^2+y^2+z^2-1
[1]=x^2+y^2-z
[2]=-x+2*y^2+z^2
Jacobian:
[0][x]=2*x^(2-1)
[0][y]=2*y^(2-1)
[0][z]=2*z^(2-1)
[1][x]=2*x^(2-1)
[1][y]=2*y^(2-1)
[1][z]=-1
[2][x]=-1
[2][y]=(2*y^(2-1))*2
[2][z]=2*z^(2-1)
Solution
Converged: True
RESULT: 0.683550745474947 0.3883237584331 0.618033988749989
EXPECT: 0.6835507455 0.3883199288 0.6180339887
SUCCESS
```

The solution converged to the specified precision of 1e-5 with only 4 iterations. Output also shows the analytical expressions of Jacobian, calculated for the solver algorithm.

Conclusions

The article introduced a simple realization of nonlinear equation system solution with analytical derivatives. The realization uses ‘ready to use’ analytical evaluation engine to provide derivatives. In the presented code simple class of the nonlinear equation system provided. This class is simple in the sense of using only double values in the formulae. This is used to explain the idea only and does not mean that the approach cannot be extended to the more general case with data of other types. For an example, the same approach was successfully applied for providing analytical capabilities for great numerical library ILNumerics.