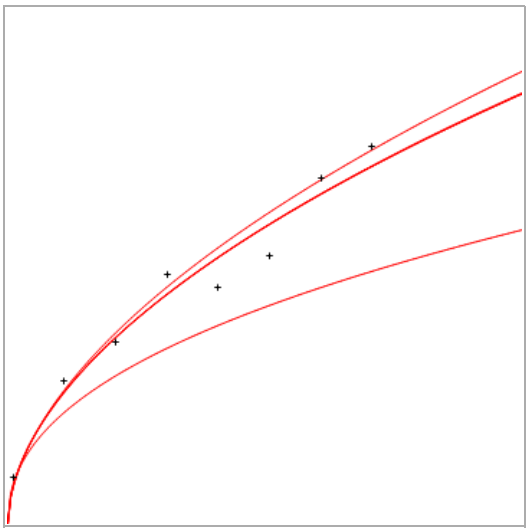
Linear And Nonlinear Least-Squares With Math.NET

*by **Libor Tinka***

中国版本 (Chinese version of this article)



CONTENTS

1. INTRODUCTION

This article explains **how to solve linear and nonlinear least-squares problems** and provides **C# implementation** for it. All the numerical methods used in the demo project stand on top of vector and matrix operations provided by the Math.NET numerical library. You can, of course, adjust the presented algorithms to work with any other numerical library that can handle commonly used matrix operations (e.g. LAPACK).

The tutorial covers **discrete least-squares problems** (i.e. fitting a set of points with known model). We will care about over-determined problems (e.g. where the number of measurements are greater than number of unknowns) and assume the problems are well-conditioned. The tutorial goes from simple to more complex and more general methods.

This article was written for programmers with basic knowledge of matrix computations and calculus.

2. LEAST-SQUARES IN GENERAL

The least-squares problem arise in many applications (statistics, financial, physics...). It can be viewed as an *optimization problem*. We gathered some data and want to find model parameters for which the model best fits the data in some sense.

The following section is purely mathematical. It is a general description of the problem, without reference to any specific application (e.g. fitting parabola to a set of points...). However, it is crucial for the following derivation of solutions for both linear and nonlinear problems.

The least-squares problem can be written compactly as:

$$\min_x \|r(x)\|^2, \tag{1}$$

where x is vector of model parameters and r is called a *residual vector* formed from individual *residuals*:

$$r(x) = (r_1(x), r_2(x), \dots r_m(x))^T.$$

Here m is a number of measurements (number of data points) and each r_j represents a discrepancy between the expected and measured value:

$$r_j(x) = y(\tilde{x}_j|x) - \tilde{y}_j.$$

As you can see, the function y is our *model function* (or *expectation model*) which return expected value for the given point $\tilde{x}_j$ and the model parameters. This value is compared to $\tilde{y}_j$, which is the observed quantity. The data points form vectors $\tilde{x}$ (e.g. points on time axis) and $\tilde{y}$ (e.g. measurements in the corresponding times). The tilde mark (~) is used to distinguish data points from model parameters (x) and the model function (y).

The residual value can be either positive or negative and in both cases it means that the model deviates from measured values. We would like to have as small residuals as possible, but in absolute value. This is why we look for least *squares* of the residuals, rather than residual values itself. Note that there exist variants of the above optimization problem, where different norms (e.g. absolute values) are used instead of squares. But least squares problem is easy to solve and have a remarkable statistical property of being so called *maximum likelihood estimator* when errors between measured values and expectations are distributed normally.

The norm in equation (1) is a 2-norm, so we can re-write this equation in terms of the *objective function* in the well-known form:

$$f(x) = \frac{1}{2}\sum_{j=1}^{m} r_j^2(x) \tag{2}$$

Of course, we would like to find parameters x for which this function reaches its minimum. The constant $\frac{1}{2}$ does not have any effect on the solution, but makes derivatives simpler (the exponent 2 of each r becomes a factor of 2, so the factor before sum disappears in the derivative).

We place partial derivatives of $f(x)$ equal to zero to find the local minimizer of $f(x)$. This is straightforward in the linear case, since the objective function happens to be convex. This means that any local minimizer of the function is also its global minimizer. In the nonlinear case, however, there can be multiple local minima and we need to exploit more of the function's structural properties.

3. LINEAR LEAST-SQUARES

Now we can be more specific about the problems we solve, but *linear least-squares* can still be applied to a whole family of functions. For example, when we are fitting straight line to 2D points (a common example), our model function has a form:

$$y(\alpha) = x_1\alpha + x_2 \tag{3}$$

But we are not limited to just straight lines. We can fit *any fuction that is linear in the paramters a*. For example, this model function:

$$y(\alpha) = x_1 \cdot \sin(3\alpha)^2 + x_2 \cdot \cos(\frac{\alpha}{2}) + x_3 \cdot 9\mathrm{e}^{\alpha}$$

can be used in linear least-squares framework as well.

The general form of the model function is:

$$y(\alpha|x) = x_1X_1(\alpha) + x_2X_2(\alpha) + \dots + x_nX_n(\alpha).$$

Here x are the model parameters, and X are *basis functions*. The basis functions can be any functions of x and can be nonlinear.

As we can observe, derivatives of residuals with respect to model parameters are particularly simple:

$$\frac{\partial r_j}{\partial x_i} = \frac{\partial}{\partial x_i}\left[y(\tilde{x}_j|x) - \tilde{y}_j\right] = X_i(\tilde{x}_j).$$

We can put all the partial derivatives to a single matrix called *design matrix*:

$$A_{ji} = X_i(\tilde{x}_j).$$

The residuals can be written as:

$$r_j = \left[\sum_{i=1}^{n} x_i X_i(\tilde{x}_j)\right] - \tilde{y}_j, \qquad j = 1 \dots m$$

or in matrix terms:

$$r = \begin{pmatrix} X_1(\tilde{x}_1) & X_2(\tilde{x}_1) & \cdots & X_n(\tilde{x}_1) \\ X_1(\tilde{x}_2) & X_2(\tilde{x}_2) & \cdots & X_n(\tilde{x}_2) \\ \vdots & \vdots & \ddots & \vdots \\ X_1(\tilde{x}_m) & X_2(\tilde{x}_m) & \cdots & X_n(\tilde{x}_m) \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} - \begin{pmatrix} \tilde{y}_1 \\ \tilde{y}_2 \\ \vdots \\ \tilde{y}_m \end{pmatrix} = Ax - y$$

3.1 SOLUTION WITH NORMAL EQUATIONS

With the above knowledge, we can write $\|r\|^2$ in terms of Jacobian:

$$\|r\|^2 = r^T r = (Ax - \tilde{y})^T (Ax - \tilde{y}) = x^T A^T A x - 2x^T A^T \tilde{y} + \tilde{y}^T \tilde{y}$$

We know from (1) that we need to minimize $\|r\|^2$ to obtain solution. This can be done by taking a derivative of the above expression with respect to a and placing it equal to zero:

$$2A^T A x - 2J^T \tilde{y} = 0$$

This is equivalent to a linear system called *normal equations*:

$$A^T A x = A^T \tilde{y} \tag{4}$$

We first compute product $A^T A$ on the left hand side, and the product $A^T \tilde{y}$ on the right hand side. Then we have a system of linear equations in a form $\mathbf{Ax = b}$.

This can be solved in various ways, so let's play with the numerical library and try different approcaches...

Since the matrix $A^T A$ is square, we can compute its inverse and obtain solution directly from equation (4):

$$x = (A^T A)^{-1} A^T \tilde{y} \tag{5}$$

The parameters can be computed this way in a single line of code:

```
x = A.Transpose().Multiply(A).Inverse().Multiply(A.Transpose().Multiply(dataY));
```

Much less expensive and more stable approach would be the Cholesky decomposition of the matrix on left hand side:

$$A^T A = \bar{R}^T \bar{R}$$

The Cholesky decomposition produces an upper triangular matrix $\bar{R}$ and its transpose so we are able to compute solution by successive forward and backward substitutions:

$$\begin{aligned} A^T A x &= A^T \tilde{y} && | A^T A = \bar{R}^T \bar{R} \\ \bar{R}^T \bar{R} x &= J^T \tilde{y} && | \text{ substitute } \bar{R}a \to \alpha,\ J^T y \to \beta \\ \bar{R}^T \alpha &= \beta && | \text{ solve for } \alpha \text{ (forward substitution)} \\ \bar{R} a &= \alpha && | \text{ solve for } a \text{ (backward substitution)} \end{aligned} .$$

Math.NET library provides means for computing both the Cholesky factorization and methods for solving linear systems of the form $\bar{R}^T \bar{R} x = J^T \tilde{y}$:

```
x = A.Transpose().Multiply(A).Cholesky().Solve(A.Transpose().Multiply(dataY));
```

The Cholesky decomposition is guaranteed to exist in our well-behaved problems (which are overdetermined - there are more points than unknowns and the Jacobian has full rank).

The presented approach (use of normal equations) works well, but only when $A^T A$ is well conditioned (i.e. it is not singular or near-singular). If A is ill-conditioned, then the product $A^T A$ will be even worse (condition number gets squared by the matrix multiplication) and we might not be able to compute solution due to roundoff errors (the computation of either matrix inverse or Cholesky decomposition may fail).

The solution with normal equations and Cholesky decomposition is useful when $m \gg n$ and it is efficient to store product $A^T A$ rather than A. It can be also effective when A is sparse.

3.2 SOLUTION WITH QR DECOMPOSITION

This approach gets over the problem with squaring the condition number by decomposing the Jacobian into product of orthogonal (Q) an upper triangular (R) matrix. Since the multiplication by orthogonal matrix preserves norm of the resulting vector, the condition number of A won't get spoiled.

We substitute $A = QR$ in equation (5) to obtain:

$$x = \left[(QR)^T QR\right]^{-1}(QR)^T\tilde{y} = R^{-1}Q^T\tilde{y}.$$

This is equivalent to linear system $QRx = \tilde{y}$ can be solved with Math.NET very easily:

```
x = A.QR().Solve(dataY);
```

If we are forced to use just basic matrix operations, we should be aware of the fact that the matrix A is rectangular, so the QR decomposition have form:

$$A = Q\begin{bmatrix} R \\ 0 \end{bmatrix} = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix}\begin{bmatrix} R \\ 0 \end{bmatrix} = Q_1 R.$$

The matrix Q would be of size $m \times m$ and the matrix R of size $m \times n$ in the original decomposition. We should split the Q and keep only first n columns (so we get Q_1 matrix of size $m \times n$. We should also cut off zero rows from $\begin{bmatrix} R \\ 0 \end{bmatrix}$ such that it will be of size $n \times n$:

```
// compute the QR decomposition
QR qr = A.QR();

// get orthogonal matrix with first n columns of Q
Matrix<double> Q1 = qr.Q.SubMatrix(0, m,0, n);
// get upper-triangular matrix of size n x n
Matrix<double> R = qr.R.SubMatrix(0, n,0, n);

x = R.Inverse().Multiply(Q1.Transpose().Multiply(dataY));
```

Of course, it is more effective to solve the linear system $Q_1 Rx = \tilde{y}$.

The solution using QR decomposition is numerically more stable than normal equations.

3.3 SOLUTION WITH SINGULAR VALUE DECOMPOSITION (SVD)

The third (and the most advanced) way of solving linear least squares problem is using SVD. This decomposition splits our matrix A into product $U\Sigma V^T$ where U, V are orthogonal and $\Sigma = \mathrm{diag}(\sigma_1, \sigma_2, \ldots, \sigma_n)$, where σ_i's are so called *singular values* with ordering: $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_n > 0$.

If the decomposed matrix is singular or near singular, we get σ_n equal to zero, or very close to zero. Furthermore, the fraction σ_1/σ_n is *condition number* of the decomposed matrix. This allows us to check for stability of our solution before we actually compute it. We can also solve *underdetermined* problems with SVD, but this is out of scope of this article (see [nocedal] or [watkins] for further reference).

Following the same logic as in the QR approach, we substitute $A = U\Sigma V^T$ in equation (5):

$$x = \left[(U\Sigma V^T)^T U\Sigma V^T\right]^{-1}(U\Sigma V^T)^T\tilde{y} = V\Sigma^{-1}U^T\tilde{y}$$

The computation of system $U\Sigma V^T x = \tilde{y}$ can be carried out by the numerical library:

```
x = A.Svd(true).Solve(dataY);
```

If we want to use just the basic matrix operations, we should be aware of the fact that J is not square:

$$A = U\begin{bmatrix} S \\ 0 \end{bmatrix}V^T = \begin{bmatrix} U_1 & U_2 \end{bmatrix}\begin{bmatrix} S \\ 0 \end{bmatrix}V^T = U_1 S V^T.$$

The matrix U would be of size $m \times m$ and the matrices Σ and V of size $n \times n$ in the original decomposition. We should split the U and keep only first n columns (so we get U_1 matrix of size $m \times n$. We should also cut off zero rows from $\begin{bmatrix} \Sigma \\ 0 \end{bmatrix}$ such that it will be of size $n \times n$:

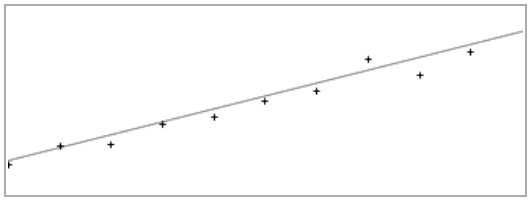
```
// compute the SVD
```

```
Svd svd = A.Svd(true);

// get matrix of left singular vectors with first n columns of U
Matrix<double> U1 = svd.U().SubMatrix(0, m,0, n);
// get matrix of singular values
Matrix<double> S =newDiagonalMatrix(n, n, svd.S().ToArray());
// get matrix of right singular vectors
Matrix<double> V = svd.VT().Transpose();

x = V.Multiply(S.Inverse()).Multiply(U1.Transpose().Multiply(dataY));
```

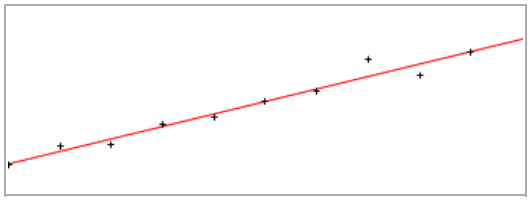
And of course, it is more efficient to solve the linear system $U_1 \Sigma V^T x = \tilde{y}$.

3.4 EXAMPLE

The plot shows a line ($y = 0.25x - 1.0$) with parameters. The data points are generated around this line:



Linear least-squares method yields a new line ($y = 0.24y - 1.1$) that minimized sum of squared vertical distances from the data points:



Most of the data points are placed below the function with true model parameters (see the gray line). The estimated line is thus placed a little lower than the original one and the estimated parametes are slightly different. We chose strongly perturbed dataset to show how errors can affect the estimated parameters. Of course, this also depends on the number of points, their relative distance and magnitude of the error introduced.

4. NONLINEAR LEAST-SQUARES

Up to now, we could solve least-squares problems directly because the model function depended on its parameters linearly. In the purely nonlinear waters, however, the parameters cannot be separated from the model function. So instead of solving linear problem $\mathbf{Ax} = \mathbf{b}$, we have something like $\mathbf{A(x)x} = \mathbf{b}$ (i.e. the design matrix also depends on model parameters).

The best we can do in nonlinear case is to guess some starting values for parameter vector and move toward solution, then update the parameters and repeat the process, hence use iterative method.

What we called the design matrix A in linear problems, would be *Jacobian* matrix $J(x)$ in nonlinear problems. It is defined the same way as the design matrix:

$$J_{j,i}(x) = \left[\frac{\partial}{\partial x_i} r_j\right],$$

but the difference is that it depends on model parameters x.

We can construct Jacobian directly from our model function (it follows from the definition):

$$J(x) = \begin{pmatrix} \frac{\partial}{\partial x_1} y(\tilde{x}_1|x) & \frac{\partial}{\partial x_2} y(\tilde{x}_1|x) & \cdots & \frac{\partial}{\partial x_n} y(\tilde{x}_1|x) \\ \frac{\partial}{\partial x_1} y(\tilde{x}_2|x) & \frac{\partial}{\partial x_2} y(\tilde{x}_2|x) & \cdots & \frac{\partial}{\partial x_n} y(\tilde{x}_2|x) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial}{\partial x_1} y(\tilde{x}_m|x) & \frac{\partial}{\partial x_2} y(\tilde{x}_n|x) & \cdots & \frac{\partial}{\partial x_n} y(\tilde{x}_m|x) \end{pmatrix}$$

In linear least squares, the partial derivatives would leave only basis functions and we would get the design matrix A. But now we have to stick with this general form since our model function cannot be broken to products of parameters and basis functions which do not depend on x.

The other difference from linear least squares it that we will need to examine not only values of the objective function, but also its first and second order derivatives. Since the objective function is $\mathbb{R}^n \to \mathbb{R}$, we will use multivariate counterparts for derivatives, namely gradient and Hessian. Both can

be expressed in terms of Jacobian:

$$\nabla f(x) = \sum_{j=1}^{m} r_j(x) \nabla r_j(x) = J(x)^T r(x),$$

$$\nabla^2 f(x) = \sum_{j=1}^{m} \nabla r_j(x) \nabla r_j(x)^T + \sum_{j=1}^{m} r_j(x) \nabla^2 r_j(x) =$$
$$= J(x)^T J(x) + \sum_{j=1}^{m} r_j(x) \nabla^2 r_j(x).$$

When solving nonlinear least squares problems, the second summation term in Hessian:

$$\nabla^2 f(x) = J(x)^T J(x) + \sum_{j=1}^{m} r_j(x) \nabla^2 r_j(x)$$

is usually negligible compared to the first term $J(x)^T J(x)$. Furthermore, the quantities $r_j(x)$ and $\nabla^2 r_j(x)$ are small near the solution (thus their product is even smaller) so we can drop the term without much harm.

This allows us to compute approximate Hessian without using second derivatives of $r(x)$. The simplified expressions for gradient and Hessian follows:

$$\nabla f(x) = J(x)^T r(x) = g(x) \tag{6}$$

$$\nabla^2 f(x) \approx J(x)^T J(x) = H(x). \tag{7}$$

This convenient, because user to provide just prescription for the model function (as in the linear least squares) plus its partial derivatives (with respect to each parameter) and the algorithm constructs all the neccessary structures by itself:

- $J(x)$ - Jacobian: Matrix of first-order derivatives of the residuals with respect to every parameter and every measurement. Approximates design matrix of the linear least squares model.
- $g(x)$ - gradient: Vector of first-order derivatives of the objective function (or current approximation to it) with respect to model parameters. Describes *slopes* of the objective function at some point x.
- $H(x)$ - Hessian: Matrix of second-order partial derivatives of the objective function (or current approximation to it) with respect to every combination of parameters. Describes *curvature* of objective function at some point x.

Every iterative method starts with an initial estimate $x_{(0)}$ (here we use index in parentheses to distinguish iteration counter from element of a vector) and compute new estimate in a *step* with the following form:

$$x_{(k+1)} = x_{(k)} + p_{(k)}$$

Forthermore, value of the objective function should be smaller at each step:

$$f(x + p) < f(x).$$

By approximating left hand side of this inequality by first-order Taylor polynomial we have:

$$\begin{gathered} f(x) + g(x)^T p < f(x) \\ g(x)^T p < 0 \end{gathered}.$$

This gives us decrease condition in terms of objective function's gradient:

$$p^T g(x) < 0. \tag{8}$$

4.1 STEEPEST DESCENT METHOD

The steepest descent method is a general function minimization method. It is not a method specialized in solving least squares problems, but it is a good one to start with.

Each iteration consists of three steps:

1. Select step direction p.
2. Select step length α (line search).
3. Update parameters with the steepest descent step $x_{k+1} = x_k + p^{SD}$, where $p^{SD} = p \cdot \alpha$.

Direction of the step (p) is always the one in which the objective function decreases most rapidly, hence the name of the method.

To derive speepest descent direction, let's start with directional derivative of our objective function f in direction p, which is

$$\nabla_p f(x) = \nabla f(x) \cdot p$$

We take the right side and continue computing the step:

$$
\begin{aligned}
\nabla f(x) \cdot p &= ||\nabla f(x)|| \cdot ||p|| \cdot \cos(\theta) && \mid \cos(\theta) = -1, ||p|| = 1 \\
\nabla f(x) \cdot p &= -||\nabla f(x)|| && \mid \cdot \nabla f(x)^{-1} \\
\nabla f(x)^{-1} \cdot \nabla f(x) \cdot p &= -\nabla f(x)^{-1} \cdot ||\nabla f(x)|| \\
p &= -\nabla f(x)^{-1} \cdot ||\nabla f(x)|| \\
p &= -\frac{\nabla f(x)}{||\nabla f(x)||^2} \cdot ||\nabla f(x)|| \\
p &= -\frac{\nabla f(x)}{||\nabla f(x)||}
\end{aligned}
$$

Note that we assume $\cos(\theta) = -1$ and $||p|| = 1$ because only angle in which the derivative is minimal is considered (the steepest direction) and we care only about direction, not magnitude of the step, so the length of the step is considered unit.

Since $\nabla f(x)$ is a vector, we can rewrite its inverse as

$$
\nabla f(x)^{-1} = \frac{\nabla f(x)}{||\nabla f(x)||^2}
$$

The steepest descent step can thus be expressed as:

$$
p_{(k)}^{SD} = -\frac{\nabla f_{(k)}(x)}{||\nabla f_{(k)}(x)||} \cdot \alpha
$$

or in terms of Jacobian and residual:

$$
p_{(k)}^{SD} = -\frac{J_{(k)}^T r_{(k)}}{||J_{(k)}^T r_{(k)}||} \cdot \alpha
$$

Obtaining the step direction is straightforward, but computing step length α is tricky, since we do not have information about scale.

The best option would be to select α such that $f(x + p \cdot \alpha)$ is minimized. This is called *exact line search*. It is possible to perform exact line search in linear case and other cases when we know the form of the model function.

In general (nonlinear) case, however, we can only perform *inexact line search*. There are several strategies on how to find α. The line search algorithms consists of setting boundaries for the α value in which the minimizer lies (bracketing phase) and then trying several values of $alpha$ to approximate the minimizer (backtracking phase). Good line search algorithm should also consider *sufficient decrease conditions*, such as Wolfe conditions to ensure that the step decreases objective function value *sufficiently* and the method will not need to run thousands of iterations to obtain reasonable precision.

To keep implementation understandable, the line search in demo implementation is reduced to just shortening the step if objective value function does not decrease, and lenghtening it oterwise. This is very ineffective, but the demo implementation of steepest descent is just a proof of concept.

4.2 GAUSS-NEWTON METHOD

To compute the iteration step p, we approximate $f(x + p)$ by second-order Taylor polynomial:

$$
f(x + p) \approx f(x) + p^T \nabla f(x) + \frac{1}{2} p^T \nabla^2 f(x) p
$$

and set derivative of the right hand side (with respect to p) to zero:

$$
\nabla^2 f(x) p + \nabla f(x) = 0
$$

From there we have formula for the so called *Newton step*:

$$
p_{(k)}^N = -\nabla^2 f_{(k)}^{-1} \nabla f_{(k)}. \tag{9}
$$

We can conclude from the above that the Newton step will take us to a *stationary point* of the quadratic approximation of $f(x)$. We cannot say whether it is a minimum (desired), maximum or a saddle point.

In either case, the Newton step should agree with the decrease condition (8) to work. We substitute this step into condition to inspect when it actually decreases function value:

$$
\begin{aligned}
p^T \nabla f(x) &< 0 \quad \mid \textbf{substitute } p \to p^N \\
\left[-\nabla^2 f(x)^{-1} \nabla f(x) \right]^T \nabla f(x) &< 0 \\
\nabla f(x) \nabla^2 f(x)^{-1} \nabla f(x) &> 0
\end{aligned}
$$

As we can see, this condition holds for any nonzero $\nabla f(x)$ (i.e. $g(x)$) only if $\nabla^2 f(x)$ (or some approximation to it, like $H(x)$) is positive definite. The positive definiteness of Hessian is an assumption in method that uses pure Newton steps.

Substituing expressions for gradient (6) and approximate Hessian (7) into (9) gives us definition of the *Gauss-Newton step*:

$$
p_{(k)}^{GN} = -H_{(k)}^{-1} g_{(k)}. \tag{10}
$$

The step can as be written as:

$$H_k p_k^{GN} = -g_k$$
$$J_k^T J_k p_k^{GN} = -J_k^T r_k$$

Compare the last equation to (4). The Gauss-Newton method actually solves system similar to normal equations in each iteration. The method actually approximates nonlinear model by the linear one and solves it. Then updates the current estimate for better approximation.

Of course, any of the presented approaches for linear least squares (Cholesky, QR, SVD) can be used to solve for Gauss-Newton step.

Being more general, Gauss-Newton method can be used to solve linear problems as well. In such case, it converges in a single iteration.

4.3 LEVENBERG-MARQUARDT METHOD

The Gauss-Newton method works well unless the Hessian gets singular or near-singular. This can happen when dealing with "highly nonlinear" functions. Other drawback of the Gauss-Newton method is when the initial guess is far from minimizer. In such case, the method converges very slowly or may not converge at all.

These problems can be addressed by modifying the Hessian matrix to improve convergence of the method. As we said earlier, the step direction is a descent direction whenever the Hessian is positive definite. This may not be the case in every point of the objective function. The function can be convex in some direction and concave in other (i.e. the Hessian is indefinite). The simple aid to *make* the hessian approximation positive definite is by adding multiple of identity:

$$H(x) + \lambda I$$

where λ is a positive number (called *damping parameter*) and I is an identity matrix of size $n \times n$. What happens is that we add positive numbers to the diagonal of $H(x)$, thus making it diagonally dominant - any negative diagonal terms become positive and any negative off-diagonal terms eventually negligible, hence the matrix can be made positive definite.

If we replace $H(x)$ in the Gauss-Newton formula (10) by our updated version with damping parameter, we obtain the *Levenberg step*:

$$p_k^L = -(H_k + \lambda I)^{-1} g_k \tag{11}$$

.

As the λ approaches zero, the method becomes more like Gauss-Newton. But the larger the λ is, however, the smaller and safer steps are made in the descent direction (now governed by $g(x)$ rather then $H(x)$).

Since λ is a scalar, it scales all diagonal elements of Hessian equally. This works generally well, if the problem is *well scaled*. Each parameter, however, can use different units (e.g. kilometers versus millimeters) and thus be scaled differently. Here comes aid from Marquardt, who noticed that diagonal elements of Hessian actually holds information about scaling. Instead of identity matrix, we can use a diagonal matrix that takes scaling into acount:

$$p_k^{LM} = -(H_k + \lambda D)^{-1} g_k$$

where

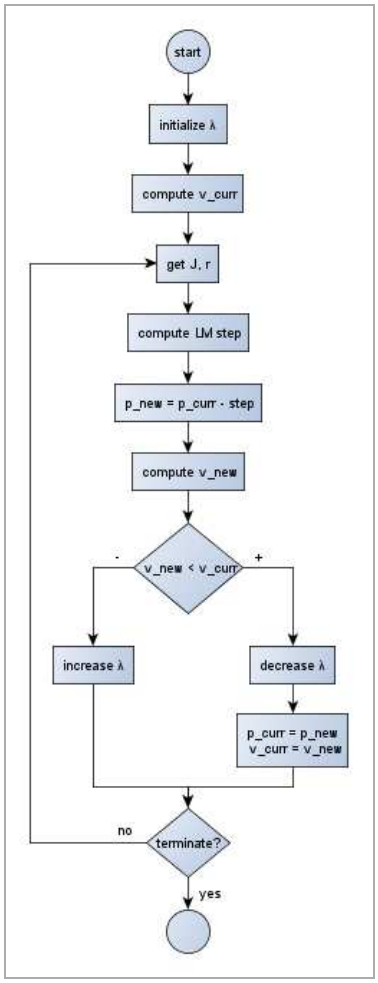
$$D = diag(H_k)$$

This is the *Levenberg-Marquardt* step used in most implementations.

We can rewrite it in terms of Jacobian and residual as

$$(H_k + \lambda D) p_k^{LM} = -g_k$$
$$\left(J_k^T J + \lambda D\right) p_k^{LM} = -J_k^T r_k$$

Here is an outline for the Levenberg-Marquardt algorithm:



The algorithm takes *initial guess* (p_curr) as an input. It initializes λ (we use 0.001 in demo project as the initial value), computes value of the objective function, obtains Jacobian and residual and computes the Levenberg-Marquardt step using all the obtained information.

Next, a new parameter vector (p_new) and objective function value (v_new) are computed.

If the objective function value has decreased (v_new < v_curr), the algorithm decreases λ (division by 10 in the demo project) and sets newly computed parameters and objective function value as the current.

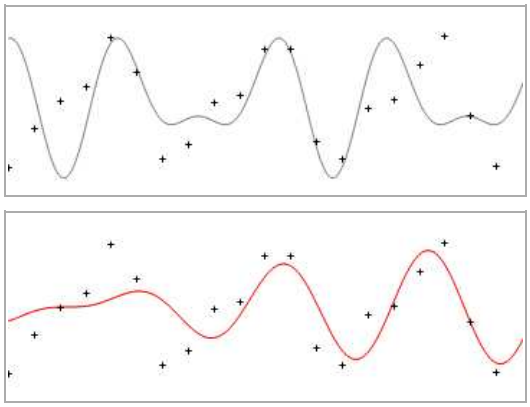
If the objective function has not decreased, the λ is increased (multiplication by 10.0 in the demo project) so that safer step in descent direction will be made in the next iteration.

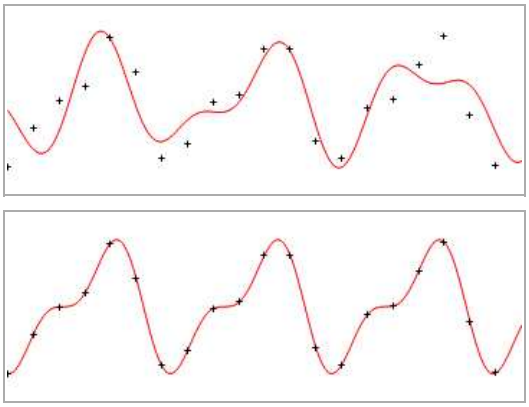
Finally, the algorithm checks termination conditions and either ends or continues with the next iteration.

4.3 EXAMPLE

We start with a model function $y = a\cos(bx) + b\sin(ax)$ where $a = 2.0$ and $b = 1.0$. The data points are scattered over the graph with variable horizontal distances. This is important since we are fitting a periodic function (bad sampling would cause unstable or ambiguous solution).

The first graph shows initial guess with $a = 1.8,\ b = 1.2$. The latter three graphs contain model function for parameters from the first, third and tenth iteration of Gauss-Newton method:



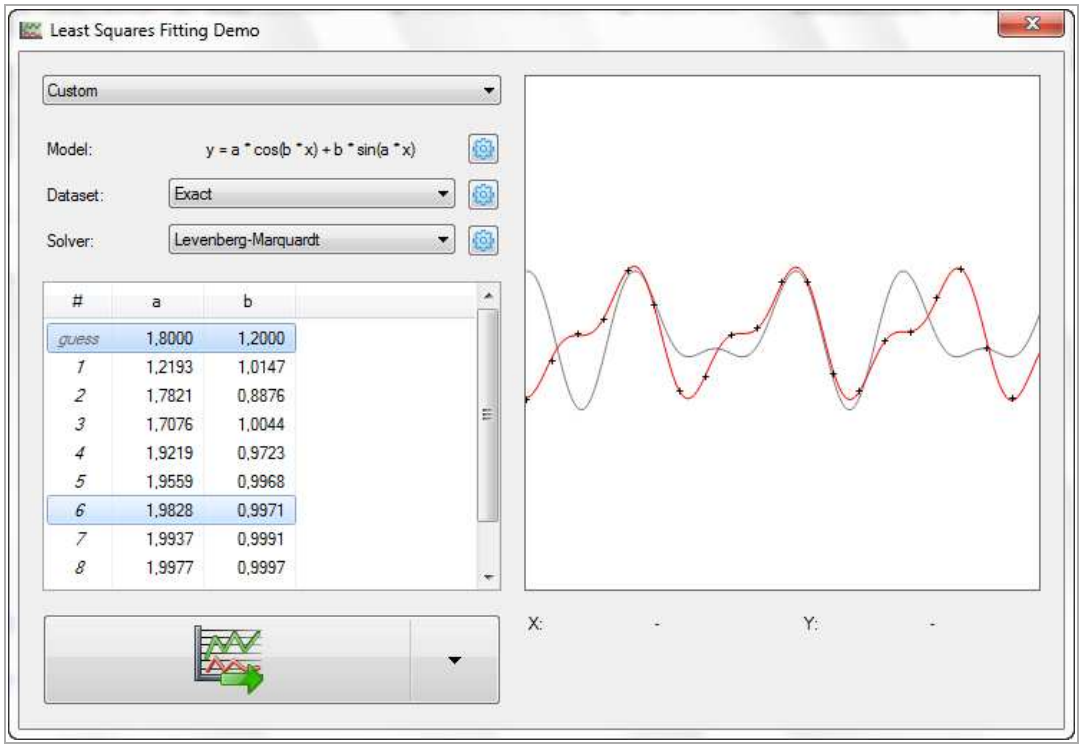


The data points are not perturbed so the estimated paramters match the true ones to four decimal places after ten iterations.

5. IMPLEMENTATION

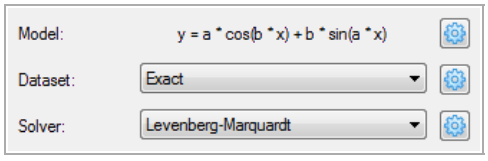
5.1 DEMO APPLICATION

All the presented methods have been implemented in a C# application called *LeastSquaresDemo*:



The combo box in the top-left corner allows you to select preset. There are several presets with sample settings.

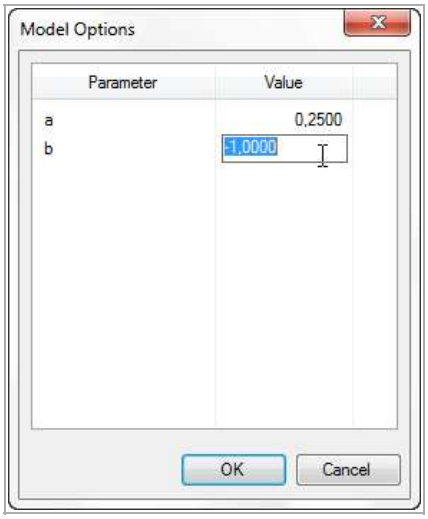
Below is the options for the current preset:



These allows you to adjust current model, dataset (set of points to fit) and the method to be used for data fitting.

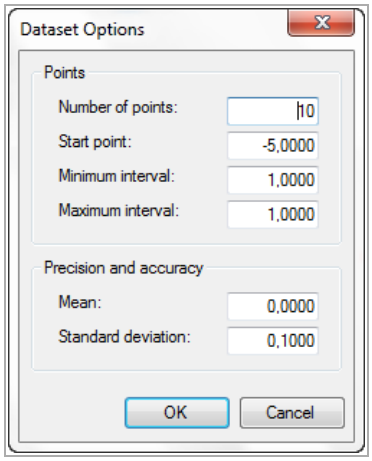
Further options are available through the "cogwheel" buttons.

The only options for model are its parameters:

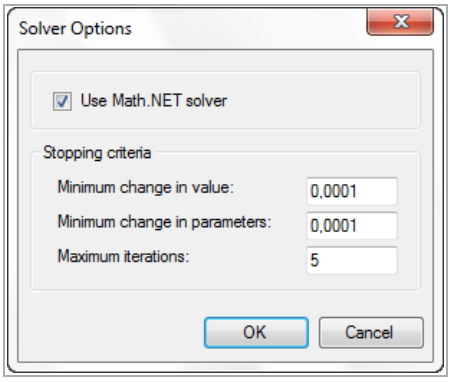


You can edit them in the dialog window. For example, the line model has two parameters that define the line.

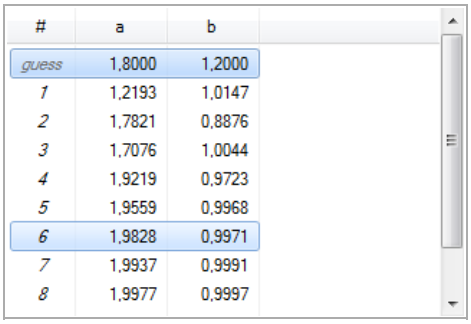
There are two kinds of datasets: *Exact* and *Perturbed*. The first one generates data points that corresponds to function values exactly. The second one adds noise to simulate real-world data. Both the horizontal and vertical positioning of data points can be further adjusted:



Finally, the method to be used for data fitting can be adjusted. You can choose whether to use effective solvers from Math.NET that use matrix decompositions (applies to linear models) and stopping conditions (applies to nonlinear models):



Under the panel with options, there is an *iteration list*:

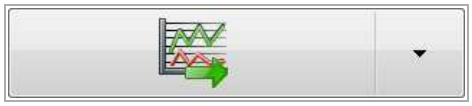


#	a	b
guess	1,8000	1,2000
1	1,2193	1,0147
2	1,7821	0,8876
3	1,7076	1,0044
4	1,9219	0,9723
5	1,9559	0,9968
6	1,9828	0,9971
7	1,9937	0,9991
8	1,9977	0,9997

It contains initial parameters (the initial guess) which you can edit by clicking on the numbers.

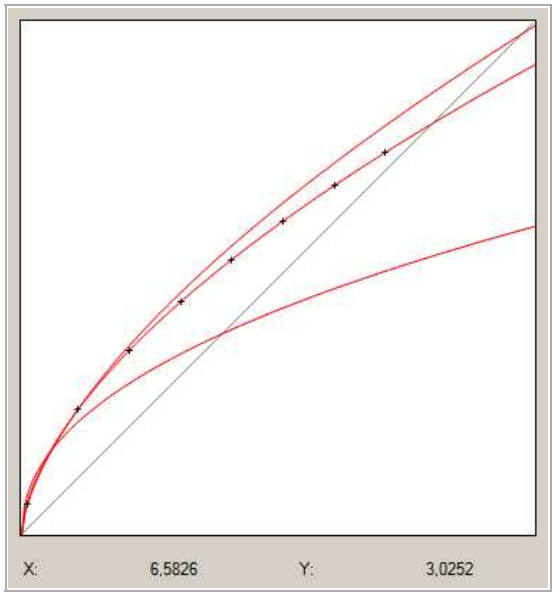
When the fitting is performed, it contains estimated parameters from every iteration of the fitting method. The linear least squares method need jsut one iteration, so there can appear only two lines (the guess which is actually not used and the solution). When nonlinear models and methods are used, there can be many iterations.

The fitting process is started by the button on the bottom left part of the form:



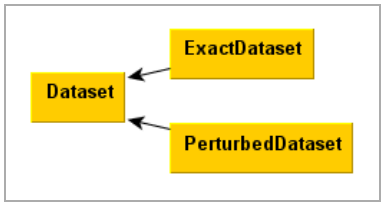
The button contains context menu so you can only fit the current data, or generate new and then fit.

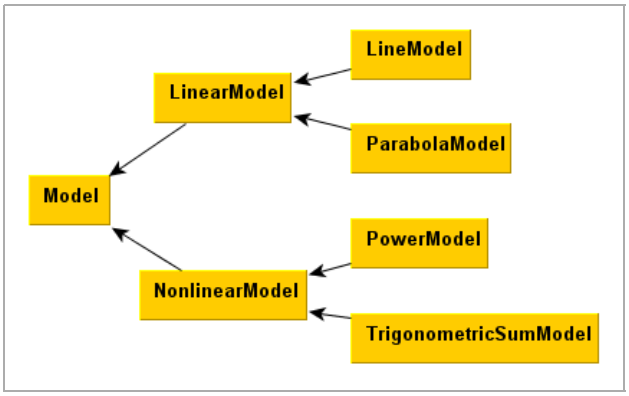
Finally, there is a graph showing the data points and all the currently selected iterations. Model function for initial guess is plotted in gray, while the computed iterations are in red:

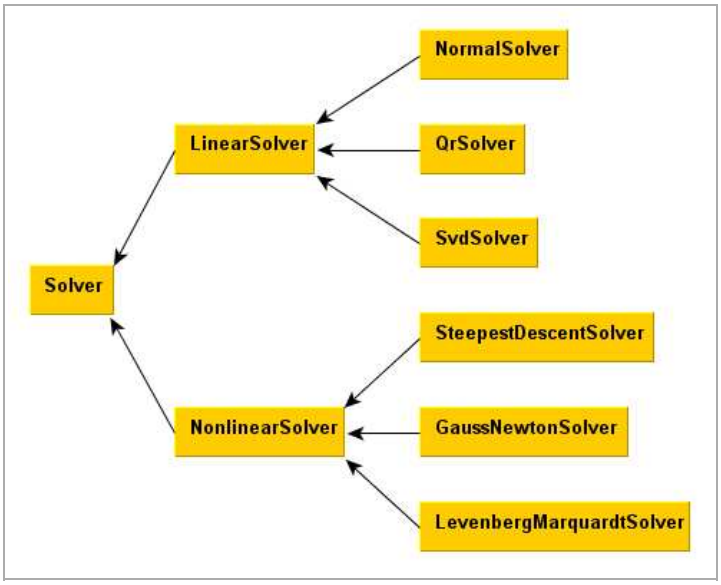


5.2 SOURCE CODE

The code regarding least squares fitting is divided into three sets of classes:







The **Dataset** class serves for generating data points, either lying on the model curve, or being scattered around it.

The **Model** class represents *model function* (or *expectation model*, if you will). There are two linear model and two non-linear. The class provides value of the model function ($y(\tilde{x}|x)$and its gradient ($\nabla y(\tilde{x}|x)$) for specified data point coordinate value $\tilde{x}$ and model parameters x.

The **Solver** class represents a method used for solving the least-squares problem, either linear or nonlinear. There are three implementations for linear least-squares and three for nonlinear least-squares.

The actual fitting is done in the **Fit()** method of **MainForm**.

The method checks whether the problem can be solved (there is enough points and we are not trying to use linear solver for a nonlinear model).

I have not implemented any extra dummy-proofing, so beware of putting in values that are out of range.

6. ALGORITHM IMPROVEMENTS

All the presented methods can be improved on multiple levels.

One of the possible improvement is the computations of function gradient when the Jacobian is expected to be sparse. Large and sparse matrices often arise in real-world application, thus I would like to point reader to [Watkins] (linear problems) or [Nocedal] (nonlinear problems) where such efficient implementations are described in depth.

Another improvement lies in better conditioning of the matrix we invert during computation. Better conditioning brings more accurate and stable solution with less iterations needed. [Watkins] covers the topic of preconditioning well.

Least squares method works best if the data points are scattered around our expectation model with normal distribution. There can also be a systematic error and we would like to know precision and accuracy of our estimator. The noisy data can also have other than normal distribution (e.g. Poisson). [Bos] covers the topic of dealing with such cases, even estimating the precision and accuracy of the expectation model is covered here.

The data can also have error in both coordinates. Such problem is discussed in [Press] (as well as the measurement of confidence intervals and other statistical properties) and in [Nocedal] (orthogonal distance regression).

7. FURTHER READING

[Bos] Van den Box, A., **"Parameter Estimation for Scientists and Engineers"**, Wiley Online Library, 2007

[Nocedal] J. Nocedal and S. J. Wright, **"Numerical Optimization"**, Springer Verlag, 1999

[Press] Press, W.H. and Flannery, B.P. and Teukolsky, S.A. and Vetterling, W.T. and others, **"Numerical Recipes"**, Cambridge Univ Press, 2007

[Watkins] Watkins, D.S., **"Fundamentals of Matrix Computations"**, Wiley, 2005

Download Sample Binary (836 KB) Download Sample Source Code (1.92 MB)