Draft: Minimal manual to `lsoptim`

Lars Eriksson
Vehicular Systems, Department of Electrical Engineering
Linköping University, SE-581 83 Linköping, SWEDEN
`larer@isy.liu.se`

May 15, 2018

Abstract

Non-linear least squares problems are one of the most common problem encountered when modeling dynamic systems. The `lsoptim` package is developed for solving non-linear unconstrained least squares optimization problems. The numerical solver is based on a variant of the Levenberg Marquardt method which is of second order and thus has good local convergence properties.

Contents

1 Tutorial

1.1 Problem formulation

We start with a set of input and output data, $\boldsymbol{x}$ resp. $\boldsymbol{y}$, where $\boldsymbol{x}$ and $\boldsymbol{y}$ are column vectors of length N in the simplest case (in the general case $\boldsymbol{x}$ can be a matrix where the different columns represent different inputs). There is a functional relationship between the components in $\boldsymbol{x}$ and $\boldsymbol{y}$

$$y_i = f(x_i, \boldsymbol{\theta})$$

where $\boldsymbol{\theta}$ is a set of M unknown parameters that we want to determine so that the sum of squared errors are as small as possible.

`lsoptim` minimizes the following function with respect to the parameters $\boldsymbol{\theta}$

$$V_N(\boldsymbol{\theta}) = \frac{1}{2}\sum_{1}^{N}(y_i - f(x_i, \boldsymbol{\theta}))^2 = \sum_{1}^{N}\epsilon_i(\boldsymbol{\theta})^2$$

where $\epsilon_i(\boldsymbol{\theta}) = y_i - f(x_i, \boldsymbol{\theta})$ is called the residual. What `lsoptim` requires is

- a function that takes the current guess of parameters as a column vector and returns the residuals in a column vector, i.e. $\boldsymbol{\epsilon} = [\epsilon_1, \ldots, \epsilon_N]^T$.
- an initial guess of the parameters $\boldsymbol{\theta}^0$ in a column vector $\boldsymbol{\theta}^0 = \left[\theta_1^0, \ldots, \theta_M^0\right]^T$.
- all initial parameter guesses must be non-zero, i.e. $\theta_i^0 \neq 0$.

Note the function `lsoptim` does not want to know anything about $\boldsymbol{x}$ and $\boldsymbol{y}$ data it only wants to know the function name and an initial guess $\boldsymbol{\theta}^0$. So in most cases it is most convenient to send the $\boldsymbol{x}$ and $\boldsymbol{y}$ data as global variables. Figure 1 illustrate how the functions interact and access the variables and should be called.

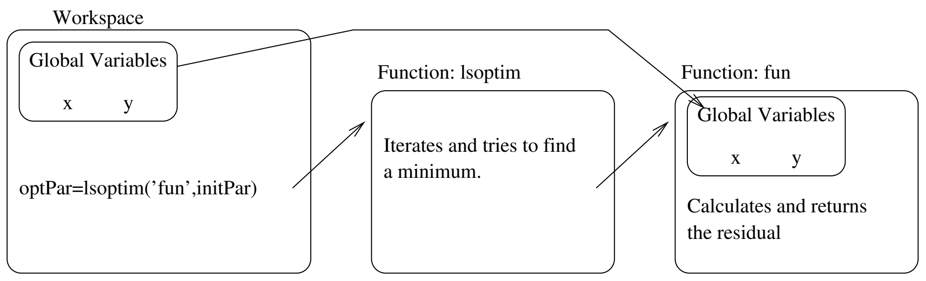

Workspace
Global Variables
x y
optPar=lsoptim('fun',initPar)
Function: lsoptim
Iterates and tries to find a minimum.
Function: fun
Global Variables
x y
Calculates and returns the residual

Figure 1: Illustration of how the functions and variables are connected to lsoptim, in the most simple case.

1.2 Practical list that outlines the procedure

The following list gives a general procedure that can be followed when fitting a model to some data using this toolbox.

1. Make the data $\boldsymbol{x}$ and $\boldsymbol{y}$ globally available.

2. Create a function (e.g. `simpleFunction`) that calculates the column vector of residuals $\boldsymbol{\epsilon}$ using the global variables $\boldsymbol{x}$ and $\boldsymbol{y}$. The residual must be a column vector.

3. Make an inital guess (e.g. `initPar`). Keep in mind that in most cases a better initial guess gives faster convergence.

4. Call `lsoptim` (e.g. `optPar=lsoptim('simpleFunction',initPar)`).

5. Evaluate how well your model fits to your data, and be aware that lsoptim only converges to a local minimum (which might be the global minimum).

1.3 Introducing the GUI

A Graphical User Interface (GUI) is designed to help manage the parameters and functions for the lsoptim package. A side effect is that when the GUI is used the parameters are normalized during the optimization. Usually the side effect is positive, since it scales all variables to the same size during the optimization, and in most cases improves the numerical properties. However there are special cases when the unintelligent scaling might make the situation worse.

There are two phases when using the GUI, a problem definition phase and a the optimization phase. For the problem definition phase there are two paths: In the first path the task is to enter function names and number of parameters and the parameter names are defined. In the second path a set of already existing parameters are defined. In the optimization phase the optimization problem is solved by changing the initial estimates and selecting different variables to make the optimization over.

The procedure for using the GUI is illustrated in Section 2.2 where the appearance of the windows is also shown.

1.3.1 Extracting information from the gui

The GUI uses a global structure called `PARAMETERS` which contains all information about the optimization problem.

1.3.2 Scaling of model parameters

The parameters are scaled to unity when the GUI is used.

2 Learning by examples

In this section a collection of some small examples of how to use the tool is given. The files for all examples are provided with the package in the subdirectory named Examples.

2.1 A small example

We have a dataset generated by a linear relation between $\boldsymbol{x}$ and $\boldsymbol{y}$, $f(x, \boldsymbol{\theta}) = \theta_1 + x\,\theta_2$ and we want to determine the parameters that fits best to the data. This example can also be solved using the normal linear least squares problem, which is the backslash operator in Matlab.

To solve the problem using the `lsoptim` package we start with the first points in the task list: making the data variables globally available.

```
global xData yData
xData=[1 2 3 4]';
yData=[2 3 4 5]'+randn(4,1)*0.1;
```

Then we write the function that returns the residual.

```
1 function [residual,out]=simpleFunction(par)
2
3 global xData yData
4
5 out=(par(1)+xData*par(2));
6 residual=yData-out;
```

Note that line 3 declares the global variables which gives access to the global variables in the work space, see Figure 1 for an illustration of how the global variables are passed to the function. The function also returns a second output, which is the output from the model. The extra output is not used by `lsoptim` but will later be used to compare the model output to the data.

Now it is time to make an initial guess then call `lsoptim`. At the Matlab prompt we write.

```
parInit=[2.9 2.9]';
parSol=lsoptim('simpleFunction',parInit);
[resid,out]=simpleFunction(parSol(:,end));
```

Finally we compare the fitted data with the actual data by plotting the data and the fitted model.

```
figure(1);clf;
```

```
plot(xData,yData,'o',xData,out)
legend('Data','Fitted Model',0)
xlabel('x-data');ylabel('y-data')
```

Figure 2 shows the resulting plot. The interested reader can also compare this solution to the solution that Matlab give when solving an over determined equation system with the `slash` operator.

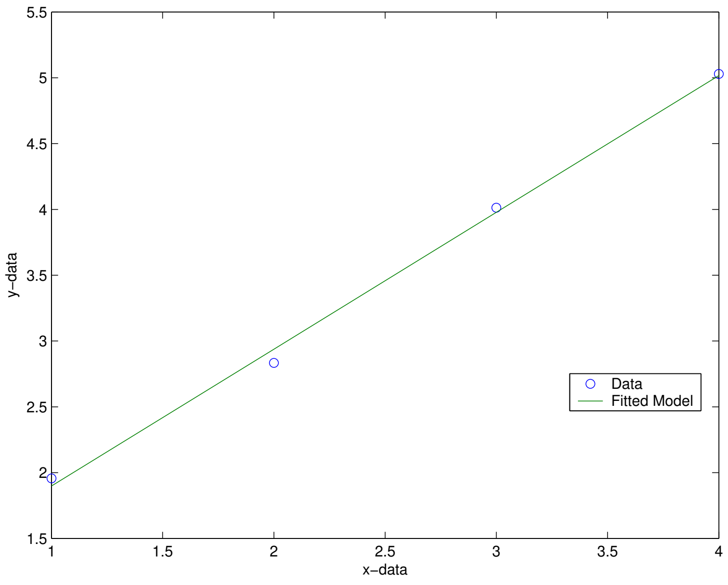


Figure 2: Plot of the data and the approximating function.

It is also possible to access the parameter trace, i.e. the route that the optimization follows during the iterations, when searching the solution. The trace is stored in a global variable named `PARAMETER_TRACE` and it can be used for diagnosis of optimization or for illustrative purposes. Figure 3 shows contours of the two dimensional least squares error $V_n(\theta_1, \theta_2)$ for the simple function together with the trace that the optimization followed. The trace is shown with red dots and lines.

2.2 A non-linear example and the GUI

This example shows how to use the GUI to setup the parameter estimation problem and how to solve it using the GUI.

Data is generated by the following input-output relation between $\boldsymbol{x}$ and $\boldsymbol{y}$,

$$y = C \cdot (\frac{\cos(x - x_0) + 1}{2} + y_0)^{-\gamma} + y_{off} \tag{1}$$

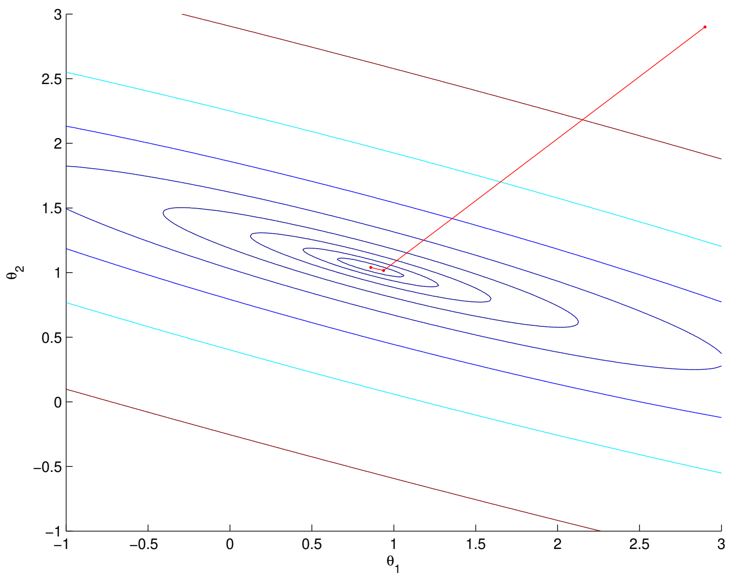


Figure 3: Contour plot of the least squares error $V_N(\theta_1, \theta_2)$ together with the parameter trace (red dots and lines).

where the unknown parameters are $\boldsymbol{\theta} = [C,\, x_0,\, y_0,\, \gamma,\, y_{off}]^{\mathrm{T}}$. first we generate and make the data globally available

```
global xData yData
xData=(-180:180)'*pi/180;
randn('state',0);
yData=1.5*((cos(xData-1.5)+1)/2+0.1).^-1.3-0.5+0.1*randn(size(xData));
```

Next we write the function that returns the residual and returns the model output.

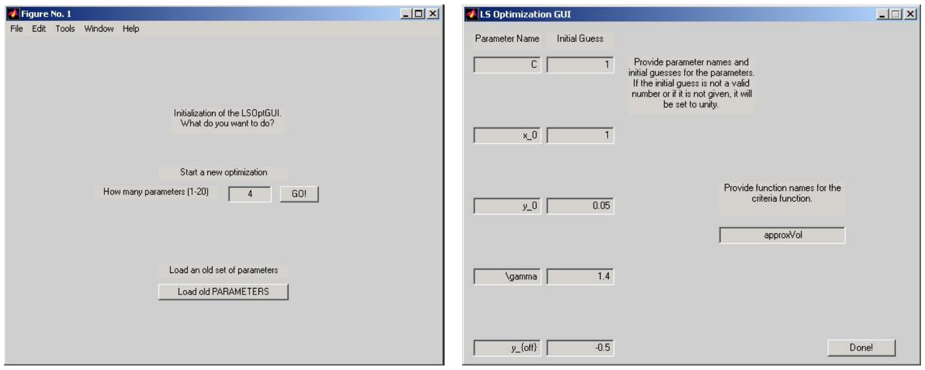
```
function [residual,out]=approxVol(theta);

global xData yData

C=theta(1);
x_0=theta(2);
y_0=theta(3);
g=theta(4);
y_off=theta(5);

out=C*((cos(xData-x_0)+1)/2+y_0).^(-g)+y_off;
residual=yData-out;
```

Now we have reached the point where we will start to solve the problem using the GUI. First we call `lsoptgui` to setup the problem. In the GUI we first enter the number of parameters in the text box, which is five for our problem, see the left picture in Figure 4. Note: It is possible to load an already existing parameter file.



Initialization of the LSOptGUI.
What do you want to do?

Start a new optimization

How many parameters (1-20) 4

Load an old set of parameters

Parameter Name	Initial Guess
C	1
x_0	1
y_0	0.05
\gamma	1.4
y_{off}	-0.5

Provide parameter names and initial guesses for the parameters. If the initial guess is not a valid number or if it is not given, it will be set to unity.

Provide function names for the criteria function.

approxVol

Figure 4: Illustration of the first two steps of setting up the gui for a particular problem. In the left figure the first window is shown, where only the number of parameters are entered. To the right the second window is shown, where parameter names and initial values are entered. It is also possible to load an already existing parameter file.

After the number of parameters is entered then press the button $\boxed{\text{GO!}}$, and a new window appears. Here the parameter names and initial guesses shall be entered. Enter the names and values shown to the right in Figure 4 and also the function name `approxVol`. Then press the button $\boxed{\text{Done!}}$ to come to the main window for the optimization phase.

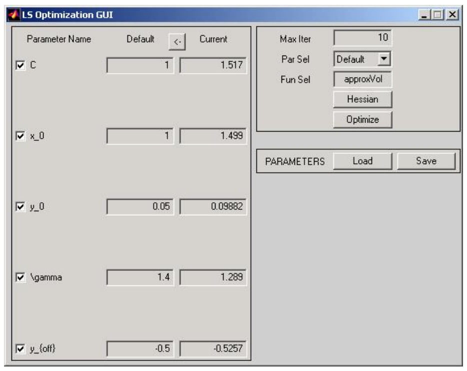
Figure 5 shows the main GUI for calling the optimization routine and that helps in investigating different initial conditions. Pressing the buttons $\boxed{\text{Optimize}}$ calls the `lsoptim` function. There are two sets of parameters Default and Current. The initial values that are sent to `lsoptim` can come from either *Default* values or the *Current* values, this selection is made using the option at *Par Sel* in the GUI. The result from the lsoptim is always put in *Current*. In the GUI there is a button $\boxed{\text{<-}}$ which copies the *Current* values to the *Default*.

The maximum number of steps that the optimization is allowed to take can also be changed, as well as the function name can be changed.

When we are finished playing with the optimization problem the parameters can be saved and later retrieved using the $\boxed{\text{Save}}$ and $\boxed{\text{Load}}$ buttons.

2.2.1 Importance of initial guess

In the default case the optimization converges after 9 iterations. To illustrate the importance of the initial guess we change the maximum number of iterations



	Parameter Name	Default	Current
☑	C	1	1.517
☑	x_0	1	1.499
☑	y_0	0.05	0.09882
☑	\gamma	1.4	1.289
☑	y_{off}	-0.5	-0.5257

Max Iter	10
Par Sel	Default
Fun Sel	approxVol

Figure 5: The main window for managing the parameters in the otimization problem. It is designed to give a possibility to quickly change initial values of the parameters to investigate the influence of initial conditions.

to 100 and the default value of y_0 from 0.05 to 0.5. Then press $\boxed{\text{Optimize}}$. Now the optimization procedure converges after 32 iterations. Finally we change y_0 to -0.5 and press $\boxed{\text{Optimize}}$. Now the optimization converges to a complex local minimum. This illustrates the following statement:
By carefully selecting initial guess. If proper care is taken the problems can be avoided and the solution to the optimization problem can faster be found.

2.2.2 Where does this problem come from?

This problem has its roots in combustion engine research where $y(x)$ represents the shape of the pressure inside the combustion chamber and where x is the crank angle. The component $\frac{\cos(x-x_0)+1}{2}$ represents a simplification of the cylinder volume and x_0 represents the angle where the volume is largest, called BDC. y_0 represents the smallest cylinder volume and y_{off} is an unknown offset in the measured cylinder pressure. C is related to the initial cylinder pressure at the maximum volume, and γ is the ratio of specific heats, i.e $\gamma = \frac{c_p}{c_v}$. The parameters listed in the parameter vector $\boldsymbol{\theta} = [C,\, x_0,\, y_0,\, \gamma,\, y_{off}]^{\mathrm{T}}$ are more or less difficult to determine by other experiments and the optimization procedure selects the parameters that gives the best fit to data.

2.3 Using linear constraints for damping

Linear constraints has been implemented from version 1.3 with the intention to dampen the steplenghts so that the model is not tested outside reasonable bounds, so that it can be used to support and damp the step length in the iterations in the solver, so that the solver does not take steps outside the allowed region. From version 1.4 the log barrier method is used to handle constraints. . . .

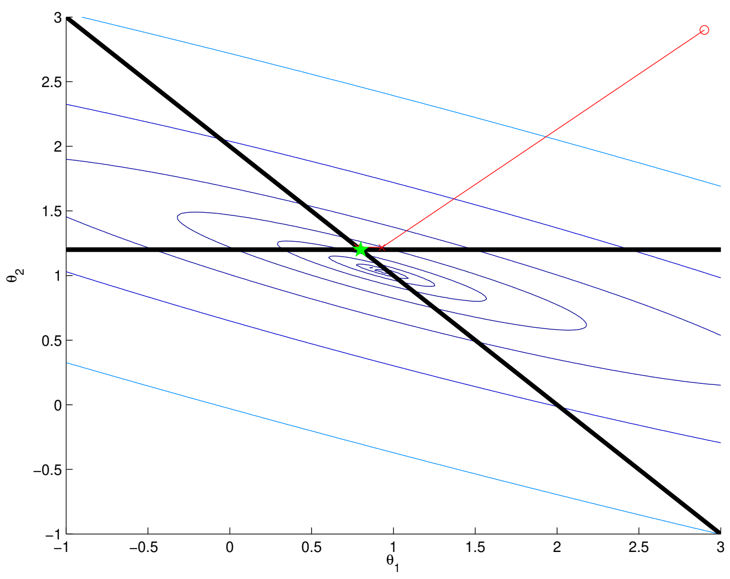


Figure 6: Illustration that constrained problems are now solved using the log barrier method. The red circle is the initial guess while the green star shows the final point.

2.3.1 Constrained problems are solved in v1.4

Constrained problems can now be solved with this method as the following example shows. We take the same problem as the simple example above but add an artificial constraint of $\theta_2 \geq 1$.

```
global xData yData
xData=[1 2 3 4]';
yData=[2 3 4 5]'+randn(4,1)*0.1;
parInit=[2.9 2.9]';
A=[0 1]; B=1.2;
parSol=lsoptim('simpFunConstr',parInit,10,[A],[B]);
```

The unconstrained global optimum is at (1,1), but the constraint forcing $x_2 \geq 1$ so that the global optimum is not attainable. The resulting trace is shown in Figure 6 With the intial guess (2.9,2.9) the optimizer now converges to (1.3548,1.2) but this is not the true optimum due to the truncation of the step. This shows that one should not use this method for solving constrained problems.

2.4 An example using Simulink to model a dynamic system

A calibration measurement has been performed on a charge amplifier, the interested reader is referred to [2] for a description of the application and the experiment. All the data and files are included in the example directory of the lsoptim package.

At $t = t_0$ a step input is applied. The dynamic response of the charge amplifier has two real poles (with time constants τ_1 and τ_2) but with unknown values and a zero in the origin, the gain K of the amplifier is also unknown and need to be calibrated.

$$G(s) = \frac{Ks}{(s\tau_1 + 1)(s\tau_2 + 1)}$$

There is also a small voltage offset U_o in the measurement system that is unknown.

A model is constructed in Simulink, see Figure 7, where the unknown parameters are included in the model as parameters that are taken from the Workspace.

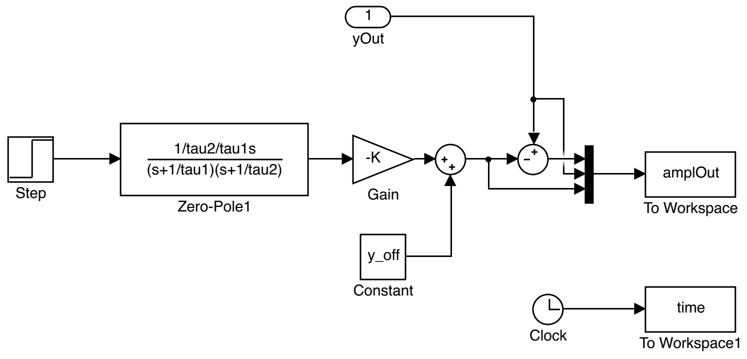


Figure 7: Simulink model for the charge amplifier with free model parameters. The measured output data is provided to the model in `yOut`. The data used for optimization is returned in the To Workspace variable `amplOut`. The simulation is controlled using the function `amplDiff.m`.

- A Simulink model that simulates the system and returns a sampled error trace. `amplSim.slx`
- A function that loads the data, sets up global variables that are needed for the simulation. `amplExample.m`

```
% Example that sets up and solves the problem for the 5mm
    charge amplifier calibration.
%

% Load the calibration data
```

```

global t yOut Ts
load amplCalib
Ts=mean(diff(t));

%  Load the parameters
global PARAMETERS
load amplCalibSetup
PARAMETERS.noOptSteps=10;

% Make variables available for Simulink during simulation

global amplOut time
global t0 tau1 tau2 K y_off Qin

figure(1);clf
lsoptgui
figure(2);clf
drawnow
amplPlot
```

- A function that simulates the model and returns the error. `amplDiff.m`

```
function yDelta=amplDiff(par);

% amplDiff(par) - calculates the difference between the 5mm
—¿ model and measurement
%
% Uses the simulink model amplSim to simulate the dynamic 5mm
—¿ system.
% Input:
%    par     - model parameters
% Output:
%    yDelta - Differece between model and data

global t yOut
global amplOut time
global t0 tau1 tau2 K y_off Qin
t0=par(1);
tau1=par(2);
tau2=par(3);
K=par(4);
y_off=par(5);

% Charge to the amplifier
U=0.104;      % Volt
C=28.0e-9;   % nF
Qin=C*U;

```

```
startStop=[0,150];
sim('amplSim',startStop,[],[t yOut]);
yDelta=amplOut(:,1);
```

- A function that can simulate the parameters and plot the result. `amplPlot.m`

```
% Simulates the model with the current parameter Values
amplDiff(PARAMETERS.current);

% Make a plot in current figure
clf
subplot(2,1,1)
plot(time,amplOut(:,1))
ylabel('Error')
subplot(2,1,2)
plot(time,amplOut(:,3),time,amplOut(:,2),'.')
ylabel('y')
legend('Model','Data','Location','SouthEast')
xlabel('Time [s]')
```

Critical details to set up the simulation for optimization:

- The optimizer needs a function that it can call this done by `amplDiff.m` that in its turn can run the simulation.
- The Simulink model and `amplDiff` needs to return a sampled signal that has the same size for all evaluations. This is generated with a $\boxed{\text{To Workspace}}$ block with sampling time set to Ts that in its turn is connected to the sampling time of the data.
- The parameters that should be changed during the iterations must be available in the main workspace. This is achieved with the global variables in the workspace and `amplDiff.m` function.

In this application the step starts some time between t=6.700 and T=6.705, but the problem is not so sensitive to this parameter, so it is first fixed to 6.7 and then let free in the GUI. The procedure to follow is:

1. Run amplExample.m
2. Press Optimize in the GUI, run a first optimization.
3. Press $\boxed{\leftarrow}$.
4. Analyze the results by running `figure(2);amplPlot`
5. Uncheck the t0 checkbox, so all parameters are optimized.
6. Press Optimze.

7. Problem is solved.

8. Analyze the results by running `figure(3);amplPlot`

The result after initial step and the final steps are shown in Figure 8, in the final results it can be seen that the model error is at the same level as the size of the discretization steps in the AD converter.

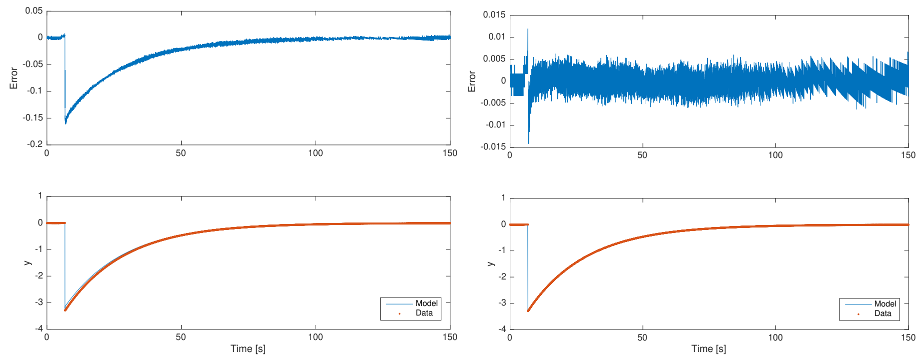


Figure 8: Left–Initial error for the charge amplifier model. Right–Final error after the optimization has been applied. The size of the error is at the same level as the discretizaion steps in the AD converter.

3 Optimization algorithm details

The minimization of $V_N(\boldsymbol{\theta})$ is a standard unconstrained non-linear least-squares problem that can be solved by several methods, see e.g. [1, 3]. In general we can say that if we can extract more information about the problem then we can use this information to solve the problem more efficiently. Methods that use the both the first and second order derivatives give faster convergence to the optimum (locally). Least squares problems have a nice structure that can be used to derive an approximation for the second order derivative of $V_N(\boldsymbol{\theta})$. This approximation can then be used to make an efficient algorithm for finding a local minimum. Here an algorithm of Levenberg-Marquardt type with a numerically computed gradient and an approximated Hessian is used.

The least squares criterion, used to select the best parameter values, can be formulated using a vector product

$$V_N(\boldsymbol{\theta}) = \frac{1}{2}\sum_{i=1}^{N}(\epsilon_i(\boldsymbol{\theta}))^2 = \frac{1}{2}\boldsymbol{\epsilon}^{\mathrm{T}}(\boldsymbol{\theta})\boldsymbol{\epsilon}(\boldsymbol{\theta})$$

From the vector product an expression for the gradient can be determined in terms of the residuals $\boldsymbol{\epsilon}$

$$\nabla V_N(\boldsymbol{\theta}) = J^{\mathrm{T}}(\boldsymbol{\theta})\boldsymbol{\epsilon}(\boldsymbol{\theta})$$

where the Jacobian, J, is defined as the gradient of the residuals

$$J(\boldsymbol{\theta}) = \nabla \boldsymbol{\epsilon} = \begin{bmatrix} \nabla \epsilon_1(\boldsymbol{\theta}) \\ \vdots \\ \nabla \epsilon_N(\boldsymbol{\theta}) \end{bmatrix}$$

The residual gradients, $\nabla \epsilon_i(\boldsymbol{\theta})$ are computed numerically using a difference approximation,

$$\frac{\partial \epsilon_i(\boldsymbol{\theta})}{\partial \theta_j} = \frac{\epsilon_i(\theta_1, \ldots, \theta_j + \Delta\theta_j, \ldots, \theta_M) - \epsilon_i(\theta_1, \ldots, \theta_j, \ldots, \theta_M)}{\Delta\theta_j}$$

where $\Delta\theta_j = 10^{-3}\theta_j$ this is the reason for why the initial parameter estimates must be non-zero, otherwise $\Delta\theta_j$ also becomes zero and the algorithm can not change that parameter value from zero.

A nice property of least squares problems is the existence of an approximation to the Hessian, which only uses the information from the Jacobian J. The Hessian is,

$$\nabla^2 V_N(\boldsymbol{\theta}) = J^{\mathrm{T}}(\boldsymbol{\theta}) J(\boldsymbol{\theta}) + \sum_{i=1}^{N} \epsilon_i(\boldsymbol{\theta}) \nabla^2 \epsilon_i(\boldsymbol{\theta})$$

and for $\boldsymbol{\theta}$ close to optimum $\epsilon_i(\boldsymbol{\theta})$ is small, which gives the following approximation of the Hessian,

$$\nabla^2 V_N(\boldsymbol{\theta}) \approx H(\boldsymbol{\theta}) = J^{\mathrm{T}}(\boldsymbol{\theta}) J(\boldsymbol{\theta})$$

3.0.1 Basis for Newton's method

The method used here is related to to Newton's method and to understand the method it is benficial to first have a look at Newton's method. To derive Newton's method we start by studying the Taylor series expansion of $V_N(\boldsymbol{\theta})$ around $\boldsymbol{\theta}^k$.

$$V_N(\boldsymbol{\theta}) \simeq \tilde{V}_N(\boldsymbol{\theta}) = V_N(\boldsymbol{\theta}^k) + \nabla V_N(\boldsymbol{\theta}^k)(\boldsymbol{\theta} - \boldsymbol{\theta}^k) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}^k)^T \nabla^2 V_N(\boldsymbol{\theta}^k)(\boldsymbol{\theta} - \boldsymbol{\theta}^k)$$

If $\nabla^2 V_N(\boldsymbol{\theta}^k) > 0$ the minimum for $\tilde{V}_N(\boldsymbol{\theta})$ is at:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}^k - [\nabla^2 V_N(\boldsymbol{\theta}^k)]^{-1} \nabla V_N(\boldsymbol{\theta}^k)$$

In Newton's method the search direction is

$$d^k = -[\nabla^2 V_N(\boldsymbol{\theta}^k)]^{-1} \nabla V_N(\boldsymbol{\theta}^k)$$

3.0.2 Optimization procedure

We start with the initial guess $\boldsymbol{\theta}^0$, and set $k = 0$. With the parameters values given at optimization step k, the next set of parameter values, $\boldsymbol{\theta}$, is updated by the following law

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k + \boldsymbol{d}^k$$

in the Levenberg-Marquardt type methods the direction, $\boldsymbol{d}^k$, is chosen as the solution to the following equation

$$(H(\boldsymbol{\theta}^k) + \nu\, I)\boldsymbol{d}^k = J^{\mathrm{T}}(\boldsymbol{\theta}^k)\boldsymbol{\epsilon}(\boldsymbol{\theta}^k)$$

With $\nu = 0$ this is the Newton method, and if $\nu \to \infty$ the direction $\boldsymbol{d}^k$ approaches the steepest descent direction. Generally it is not guaranteed that $\boldsymbol{d}^k$ is a descent direction, for example if the problem is very non-linear or if the residuals, ϵ_i, are large. Therefore, the parameter ν is varied during the optimization to ensure that a descent direction is received. If $V_N(\boldsymbol{\theta}^{k+1}) \geq V_N(\boldsymbol{\theta}^k)$ then ν is increased and new values of $\boldsymbol{d}^k$ and $\boldsymbol{\theta}^{k+1}$ are computed. This is repeated until a descent direction is received. To extend this simple method further with a linear search has not been necessary since the procedure usually results in descent direction.

The stopping criterion for the search algorithm has been chosen as

$$\text{If } V_N(\boldsymbol{\theta}^k) - V_N(\boldsymbol{\theta}^{k+1}) < \xi V_N(\boldsymbol{\theta}^k) \text{ then stop}$$

where ξ is a chosen to be a small number, now 10^{-5}. This means that the last iteration did not improve the objective function more than a certain degree.

In many engineering applications the model parameters can range over several decades. When using the GUI the optimization is performed using a set of scaled variables. A linear transformation $\hat{\boldsymbol{\theta}} = D\boldsymbol{\theta}$ with a diagonal matrix, where the diagonal elements are $D_{i,i} = 1/\theta_i$, is applied to the initial guess so that the optimization procedure starts with a set of variables $\hat{\boldsymbol{\theta}}$ that are all equal to 1.

When this method was applied to estimation of parameters for cylinder pressure models it usually converged after 3 to 12 iterations, depending on how many and what parameters were included in the optimization. For example, if the model is over-parameterized then the Hessian is close to singular which makes it numerically difficult to invert. In this case the objective function does not decrease as fast as for a well posed problem, during the opimization.

References

[1] Åke Björk. *Numerical Methods for Least Squares Problems.* SIAM, 1996.

[2] Lars Eriksson. *Spark Advance Modeling and Control.* PhD thesis, Linköping University, May 1999. ISBN 91-7219-479-0, ISSN 0345-7524.

[3] David G. Luenberger. *Linear and Nonlinear Programming.* Addison Wesley, 1984.