

## Case Study on Quasi-Newton Optimization of RF Matching Networks

### Networks

In this separate document, we would to do some more detailed investigations on optimization using the RealTime app for matching networks:

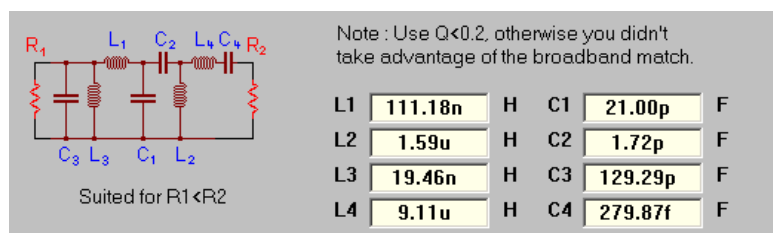
- Follow typical user or beginner questions around optimization
- See the impact of goal function definition changes
- Impact of accuracy of the gradient approximations and internal optimizer parameters on speed and accuracy
- Try out some often proposed speed-up techniques.

Due to benchmarks like those from Box's, BFGS is among the best optimizers, so it is part of many optimization packages and modern design environments. For instance, Nelder-Mead—a popular simpler good optimizer—becomes *worse* at more complex problems. That and many other optimization aspects can be checked interactively in real time with our program complementing the book, e.g., with the optimization for different complex high-Q broadband matching networks.

Actually this pdf is quite long, but on the other hand this is only the top of an iceberg of many other experiments you and software developers do!

The network we inspect now is not simple at all, formulas to hand-calculate the elements exist, but are accurate only for low impedance ratios  $R_2/R_1$  up to app. 5; here we choose 175! Also note that this network is very tough to design with manual techniques like a Smith chart, and simpler circuits are only suited for much more narrow-band matchings. The optimization criteria are to get a flat passband behavior, like a Butterworth filter.

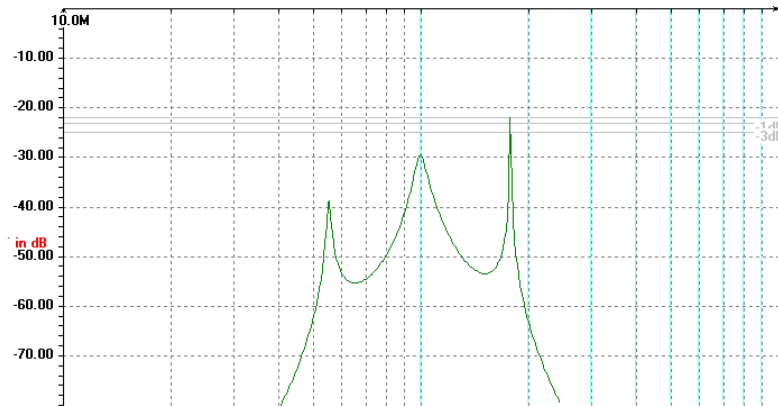
Passive 8-element LC filter in broadband bandpass topology: optimized values shown



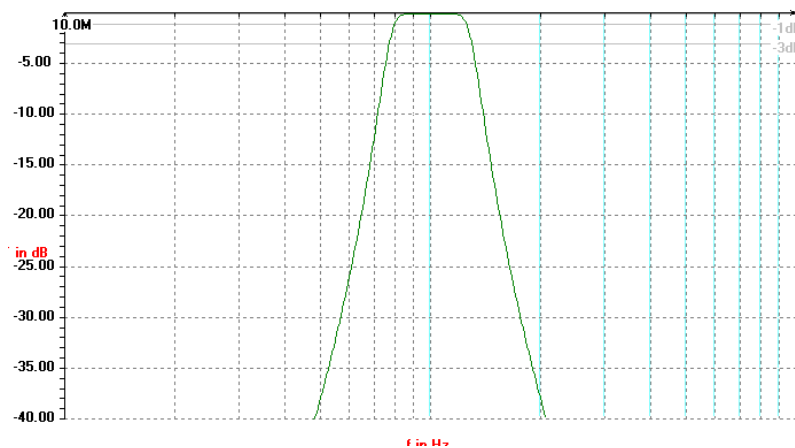
$R_1 = 20 \Omega$ ,  $R_2 = 3500 \Omega$ , start values taken from manual calculations, which are off by up to 10×:

$L1 = 210.54 \text{ nH}$ ,  $L2 = 842.17 \text{ nH}$ ,  $L3 = 10.94 \text{ nH}$ ,  $L4 = 81.24 \text{ uH}$   
 $C1 = 35.81 \text{ pF}$ ,  $C2 = 1.00 \text{ pF}$ ,  $C3 = 231.60 \text{ pF}$ ,  $C4 = 31.18 \text{ fF}$

Response with start values:



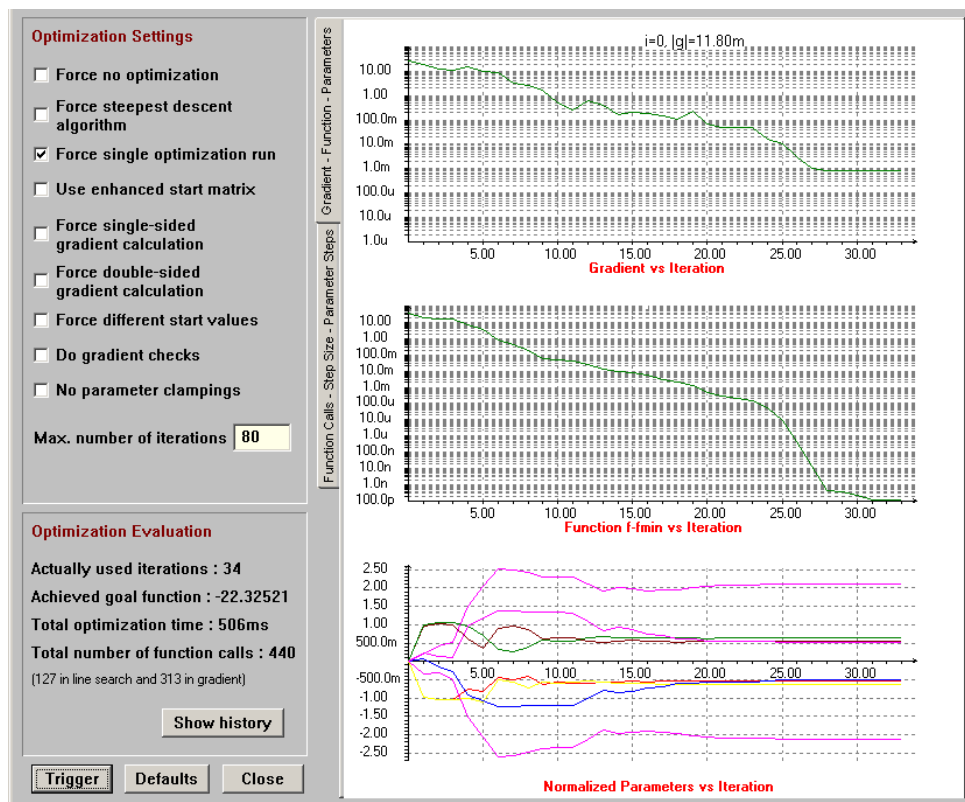
Response with final values:



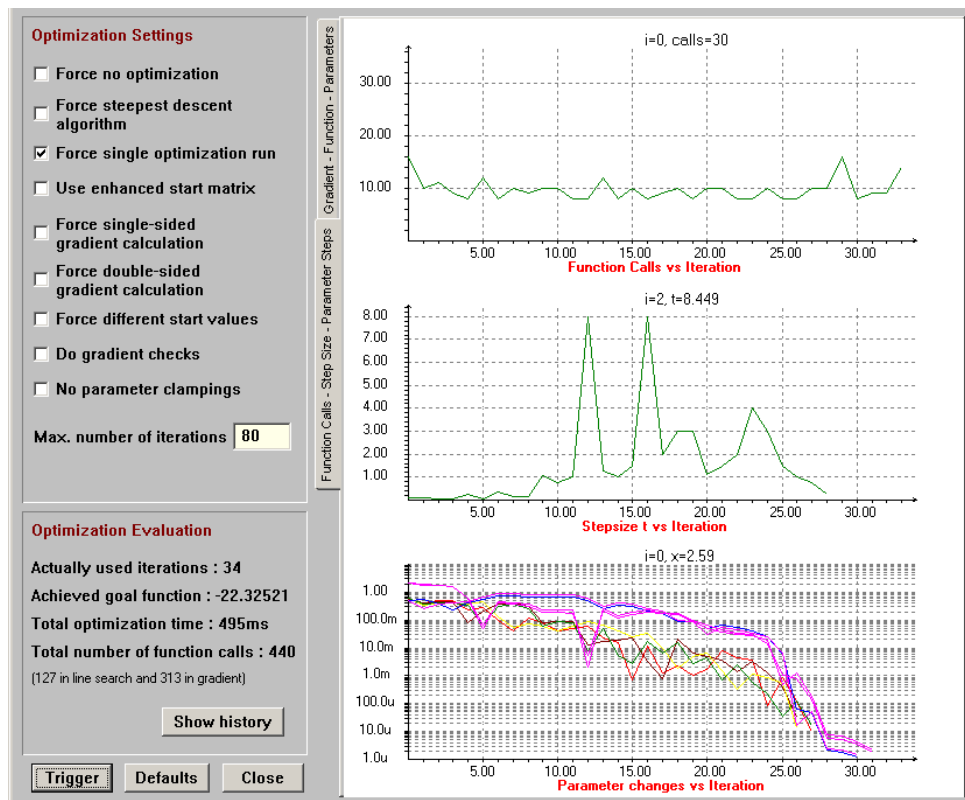
The optimization task involves a highly nonlinear goal function (although the circuit itself is linear) and the start parameters used here as default (calculated by hand) are quite bad.

Interestingly a moderately good frequency response can be already achieved with still quite non-optimum parameter sets with parameters off by 20% or much more. This means the optimum is quite flat.

Our software to the book ANPASS uses BFGS internally, it needs only app. 300 goal function calls (and 25 iterations) for obtaining a good parameter set, and app. 440 to 600 calls to really terminate with highest accuracy after 32–35 iterations.



Gradient, goal function and normalized parameters vs. BFGS iterations.



Effort in each iteration, stepsize and parameter changes from iteration  $i$  to  $i + 1$ .

Also, the gradient and inverse Hessian matrix  $H^{-1}$  is monitored over the iterations in ANPASS. This allows further inspections.

## Gradient Behavior and Hessian Matrix

The gradient changes of course a lot by many orders of magnitude (e.g., from 10 down to  $1e-6$ ). However, as from basic theory the goal function  $f-f_{\min}$  itself changes typically even more decades (due to its quadratic behavior). For highly nonlinear problems and at few moments also in the LC filter optimization it could be that  $|G|$  does not decrease monotonically, and only  $f$  has to do so for a local optimizer.

Any normal gradient optimization will usually not go in directions with zero sensitivity, so it is already doing a kind of classification internally. This only takes a function evaluation, i.e., a bit time. Adding variables with zero sensitivity will only slow down optimization a bit, but does not degrade convergence and accuracy.

Example with one variable with zero sensitivity: The optimizer is clever enough not to change  $x_4$ !

Iteration 1 (13 calls) :

Current goal function and gradient :  $f = -1.567$ , reduction  $df = 0.92$ ,  $|G| = 116.58$

X : -0.437909813 -0.388333726 0.413160078 0.000000000

Gradient : 37.823008403 59.896137827 92.584199687 **0.000000000**

Search direction : -0.437909813 -0.388333726 0.413160078 **0.000000000**

and

Final X vector : -0.153687379 -0.112623282 0.053428087 **0.000000000**

Compared to start X : 0.000000000 0.000000000 0.000000000 **0.000000000**

X(start)-X(final) : 0.153687 0.112623 -0.053428 **0.000000**

The Hessian matrix or its inverse  $H^{-1}$  changes not so much after app. 10–20 iterations anymore (in opposite to the gradient). On the other hand, at the beginning  $H^{-1}$  is *much* different and even *not positive-definite*, e.g., some  $df^2/dx^2$  and some eigenvalues are *significantly* negative. This is one reason to prefer quasi-Newton algorithms over pure Newton. Typically the real  $H^{-1}$  can be quite often non-positive-definite, but BFGS makes sure that the *iteration matrix* is positive-definite in most cases. Therefore also using H information at start-up will not speed-up optimization, as it does for simpler tasks with near-quadratic behavior.

At the end of the optimization, the real  $H^{-1}$  and the BFGS approximation are very close, which indicates that the linear searches and the gradient calculations are quite accurate enough.

Looking at the individual matrix elements one can find that also most entries *out of the diagonal* are quite large, i.e., there is a lot of *correlation* between the parameters.

Example data: Start values :  $f = 17.533$ ,  $|G| = 25.77$ ,  $X = 0$

Real Hessian Matrix H at start :

```
-5.647927 -0.354676 -5.033360 -0.119353  2.596412  2.343032  0.007292  0.007443
 0.067045 -6.214830  0.218109 -6.220012 -0.155931 -0.140293  1.089999  0.903702
 0.934581  0.127290 -0.971607  0.524625 -1.684655 -1.525275 -0.125755 -0.105371
 0.198756  0.893468 -0.464037 -0.414063 -1.108320 -1.002618 -0.758818 -0.682397
-0.127844 -0.000774 -2.010170 -2.542264 21.321453 19.821812 -2.181424 -1.946230
-0.114488 -0.000789 -1.818579 -2.299881  0.929665  2.683638  0.054595  0.048544
-0.003507 -0.044027  0.000779 -1.970261 -0.102311  0.020344 54.058338 53.833464
-0.002942 -0.028056  0.001284 -1.759802 -0.091280  0.018089  0.995840  1.683782
```

Real inverse Hessian Matrix iH at start :

```
 1.931256  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000
 1.515239  2.799346  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000
-2.156935 -1.765070  2.222883  0.000000  0.000000  0.000000  0.000000  0.000000
-1.587212 -2.850842  1.848890  2.745429  0.000000  0.000000  0.000000  0.000000
 0.180967  0.147982 -0.180587 -0.155022  0.369322  0.000000  0.000000  0.000000
-0.030333 -0.024824  0.029903  0.026030 -0.346482  0.372639  0.000000  0.000000
 0.098887  0.176935 -0.115381 -0.160515  0.009708 -0.001660  0.607469  0.000000
-0.076238 -0.136219  0.089006  0.122030 -0.007497  0.001289 -0.591430  0.593901
```

Eigenvalues of real start iH : 8.399330353 1.514076615 1.170014388 0.704779079 0.019258561  
0.008964922 -0.078440898 -0.095738348

Eigenvalues are partly negative, also some fxx!

End of optimization : Iteration 30 (373 calls) :  $f = -22.325$ ,  $|G| = 0.54m$

X : -0.537254133 0.547179228 -0.634096754 0.633972511 -0.511028761 0.509010962 2.123356805 -2.121750199  
G : 0.000197549 0.000235969 0.000283648 0.000265968 -0.000120535 -0.000159126 -0.000059002 -0.000087998  
S : 0.000034971 0.000001125 -0.000089482 -0.000071916 -0.000299762 0.000275109 -0.000494417 0.000369723  
Angle between -G and S (in degrees) : 79.34

Current real Hessian Matrix H :

```
39.387178 -15.030981 34.933709 -12.158985 -10.822108 -9.311982 -2.730749 -4.171907
-15.030981 39.391859 -12.145014 34.933052 -2.733884 -4.183125 -10.824479 -9.313279
34.933709 -12.145014 38.733567 -14.510668 -10.828884 -10.547625 -2.692069 -3.026148
-12.158985 34.933052 -14.510668 38.729642 -2.696687 -3.038787 -10.826184 -10.542998
-10.822108 -2.733884 -10.828884 -2.696687 6.946523 6.775682 2.043681 2.207772
-9.311982 -4.183125 -10.547625 -3.038787 6.775682 6.995009 2.208180 2.051088
-2.730749 -10.824479 -2.692069 -10.826184 2.043681 2.208180 6.937411 6.764367
-4.171907 -9.313279 -3.026148 -10.542998 2.207772 2.051088 6.764367 6.980736
```

Current real inverse Hessian Matrix :

```
 0.326578  0.106585 -0.231305 -0.046097  0.800937 -0.681120  0.046499  0.069242
 0.106585  0.326727 -0.045898 -0.231063  0.046522  0.069225  0.803496 -0.682919
-0.231305 -0.045898  0.310675  0.129522 -0.498153  0.639421  0.095188  0.008261
-0.046097 -0.231063  0.129522  0.310375  0.091364  0.011898 -0.499337  0.640555
 0.800937  0.046522 -0.498153  0.091364 10.257554 -9.773200  6.130830 -5.850580
-0.681120  0.069225  0.639421  0.011898 -9.773200  9.845678 -5.842020  5.839467
 0.046499  0.803496  0.095188 -0.499337  6.130830 -5.842020 10.277712 -9.794738
 0.069242 -0.682919  0.008261  0.640555 -5.850580  5.839467 -9.794738  9.870264
```

Current inverse Hessian Matrix from BFGS :

```
 0.269023  0.081496 -0.210689 -0.059930  0.085868 -0.011059 -0.575731  0.648033
 0.081496  0.317212 -0.029802 -0.231954 -0.344725  0.430472  0.523989 -0.408351
```

```

-0.210689 -0.029802  0.351044  0.182048  0.270427 -0.078758  0.950721 -0.753452
-0.059930 -0.231954  0.182048  0.359566  0.686704 -0.541004  0.207559 -0.001692
 0.085868 -0.344725  0.270427  0.686704  4.294090 -4.067071  1.915950 -1.002862
-0.011059  0.430472 -0.078758 -0.541004 -4.067071  4.412566 -1.746729  1.129591
-0.575731  0.523989  0.950721  0.207559  1.915950 -1.746729  8.179361 -6.764037
 0.648033 -0.408351 -0.753452 -0.001692 -1.002862  1.129591 -6.764037  5.993818

Current eigenvalues of real iH : 31.734557547  8.254865168  0.667068407  0.551772939  0.173130101  0.118668380  0.015807497
0.009692377
Current eigenvalues of BFGS iH : 15.385342016  7.371443038  0.647647570  0.520917403  0.156084723  0.068429042  0.017114348
0.009702694

```

In most implementations, the identity matrix is used as BFGS start matrix, but that might be non-optimum. Some experiments have been done on this. Generally a more clever start matrix, e.g. exploiting the gradient calculations, can give a speed-up, but sometimes also the convergence is slightly worse.

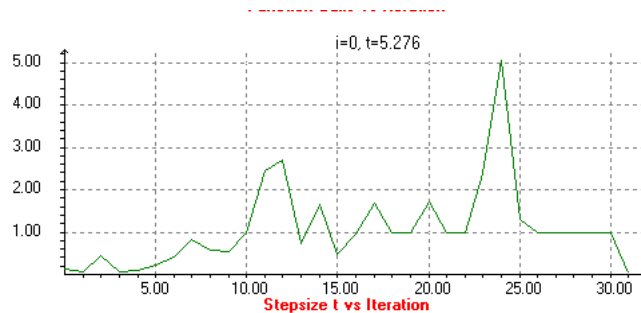
Some articles recommend to reset the BFGS iteration matrix frequently, to get rid of too old information. One example could be if the normalized stepsize  $t$  was big in the previous iteration. Then a reset makes sense, because after a *big* step we might go to *another* region with *other* behavior of the goal function.

To check of such resettings would impact the optimization speed significantly, and we have monitored  $t$  in a difficult optimization.

*Remark: Also usual BFGS implementations feature often some resetting for such reasons, and switch back to steepest gradient which would always lead to a down-hill direction, even if the Hessian is not well-behaving. In ANPASS a reset is done if  $|SG| > -\alpha \sqrt{(|GG| * |SS|)}$ .*

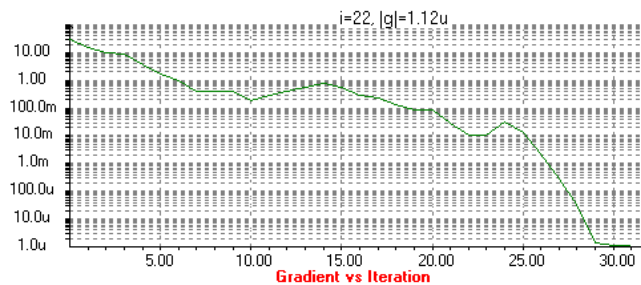
LC network with  $R_L = 5 \text{ k}\Omega$ ,  $R_G = 50 \Omega$ :

Stepsize vs. iteration:



A reset would be triggered only 2 times: at #11/12 iteration and at #24.

Interestingly these are the points in which the magnitude of the gradient goes *not* down:



It seems that such reset conditions happen frequently, like after 10 iterations in average. For highly nonlinear optimization problems (like Rosenbrock) or problems with fewer parameters, it happens more frequently.

All quasi-Newton optimization algorithms are similar. DFP and BFGS are most popular. DFP formula to update the inverse Hessian is a bit older and many numerical papers claim that BFGS is a bit better. That can be confirmed here too. Generally it can easily happen with DFP that the iteration matrix becomes non-positive definite, which results in bad search directions. The typical BFGS speed advantage is in the range of 20% in this problem.

It was also tried to use the Hessian *directly from start* of BFGS, i.e., BFGS starts then not with  $S = -G$  as usual search direction, but takes also information from  $d^2f/dx dy$  into account from direct H pre-calculations. This takes additional time due to function evaluations. (+28 for  $n = 8$ ). If  $H^{-1}$  has negative eigenvalues, then  $H^{-1}[i,i]$  has been increased accordingly for damping. This way BFGS acts still close to a true Newton algorithm and usually a bit faster than with not taking advantage of  $2^{\text{nd}}$  order derivatives. Biggest speed-up (by app. 50%) has been found for moderately complex optimization problems (medium resistor ratio for broadband match), because small optimization problems are not hard anyway and for too nonlinear problems the gain in using  $H^{-1}$  is not big because large damping is

needed forcing to  $S = -G$ . On the other hand, the angle between used  $S$  and  $-G$  can be quite large (e.g., 75 degrees); this explains why BFGS is usually *much faster* than simple steepest descent ( $S$  fix to  $-G$ ).

## Parameterization Variants

Direct optimization of the LC element values would be very difficult, because L and C can differ by orders of magnitude. ANPASS therefore transforms the parameters by using the logarithm of the normalized element values, so the start value of X is always zero, due to:

```
Xs[1] = C1           // storing start element value
X[1] = 0
later
C1 = Xs [1]* exp(X[1]);    // denormalization for simulation
```

There are also transformations possible like simply normalization via division by the start value. Many other optimizers are doing this:

```
Xs[1] = C1           // storing start element value
X[1] = 1
later
C1 = Xs [1]*X[1];    // denormalization for simulation
```

However, that is not as good as taking the log, and the number of iterations to the optimum increases from 36 to 62! So the question arises, if there are even better transformations, then the log is possible? The answer is clearly yes, although making it better is harder than making it worth.

One can use the LC ratios (setting often the filter Q-factor) and the LC products (setting filter resonance frequencies) as variable, again with log, this gives a (quite moderate) 10% speed-up:

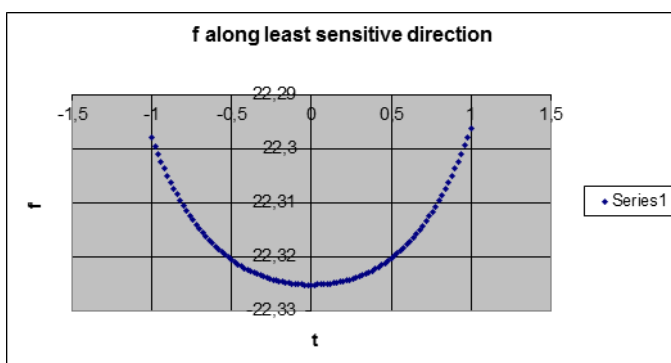
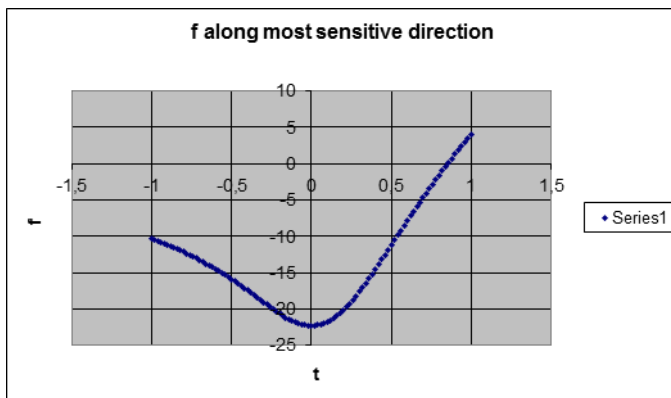
```
Xs[1] = L1/C1 // storing start element value
X[1] = 0
Xs[2] = L1*C1 // storing start element value
X[1] = 0

L1byC1 = Xs[1]* exp(X[1]);
L1C1 = Xs[2]*X[2]);
L1 = sqrt(L1byC1*L1C1);
C1 = L1C1/L1;
```

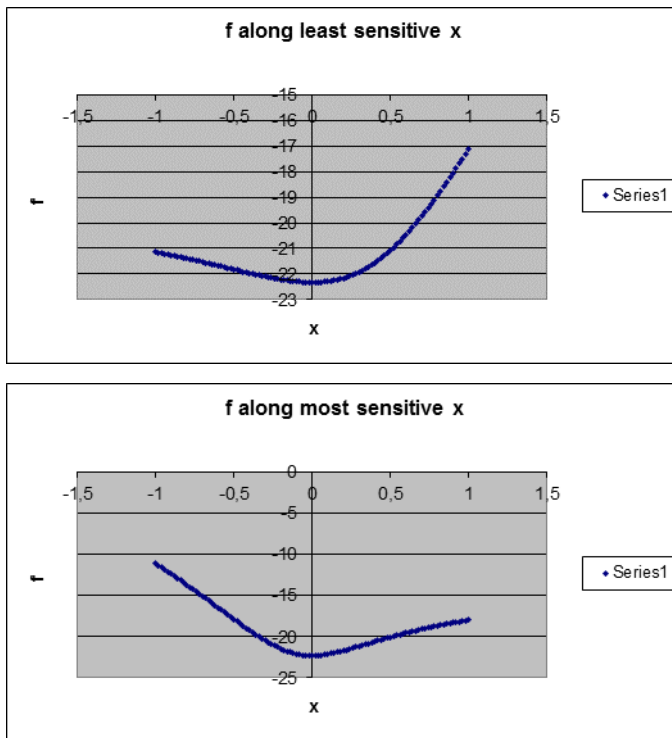
Note: For transistor circuits, such transform is also useful, e.g., one can first optimize W/L (e.g., to set  $g_m$  or  $V_{DSAT}$ ) and later W\*L (to optimize on area and matching).

If ANPASS optimizes with *different start* parameters (detuning by  $1.6\times$ ), then BFGS is able to optimize almost accurately, so the final parameters only differ by 0.00002%, but that is *not* trivial. At the end all  $f_{xx}$  are positive and in the same order, also all eigenvalues are positive, i.e., if you look into the neighborhood of the optimum, then  $f$  behaves as a quadratic function really having a well-defined optimum (that is not the case at the starting point!) Having  $f_{xx} = a$  (app. 30) means that  $f \approx f_0 + \frac{1}{2}f_{xx}x^2$ , so accepting  $\Delta f = 0.01$  (dB) off leads to *parameter inaccuracy*  $\Delta x = \sqrt{(2\Delta f/f_{xx})} = 0.025 \approx 2.5\%$ .

This calculation is only valid if *all other*  $x_i$  are fix. The existence of some small eigenvalues indicates that the parameters are not really well defined as the big  $f_{xx}$  may indicate. Doing the calculation with worst-case eigenvalue instead of  $f_{xx}$  leads to much larger  $\Delta x$ , i.e., to get *really* accurate parameters an optimization with very *high accuracy*  $\Delta f$  is needed! Internally ANPASS performs an eigenvalue analysis and makes a parameter sweep along the eigenvector with smallest and largest eigenvalue. Interestingly—as the ratio of eigenvalues indicates—the sensitivities along these two most extreme eigenvectors are highly different. Along *most* sensitive direction, a 2.5% change is equivalent to a change of app. 0.03 (dB) in  $f$ . Along the *least* sensitive direction 0.03 dB are equivalent to a parameter change by  $2.7\times$ !



The next pictures show  $f$  as a function of the most and least sensitive *parameter*. The difference is much smaller compared to walking along the extreme eigenvectors, i.e., the picture tells only a *small part of the real story*:



Parameter correlation is no surprise in most filter optimizations. Indeed for many filter optimization tasks the value of the diagonal Hessian matrix entries is similar to the non-diagonal elements. In more complex examples however, the user would typically include also parameters on which the impact on goal functions is not so clear and often *minor*, so that the non-diagonal elements become smaller. Also really large optimization problems tend to have a sparse Hessian matrix.

For transistor sizing on  $W$  &  $L$  we can also expect tight correlations, especially between *neighboring* parameters, i.e.,  $H$  will often have a sparse band structure (if parameters are sorted well).

## Points in Gradient Calculations and Line Search

Any gradient optimizer spends time mostly in gradient calculations and in line search (not in internal calculations!). Using *single-sided* gradient approximation  $n + 1$  function calls is needed; for more accurate *double-sided* formula, it is  $2n + 1$ .

The number of points taken in linear search is not so easy to estimate. For a Newton-algorithm and well-behaved  $f$  it might be one single call, for more nonlinear problems it can high, e.g., 30. Such extreme values are typical mainly at the beginning (e.g., due to bad start parameters) or at the end (e.g., due to numerical noise becoming important due to small gradients).

So generally many gradient optimizers spend *more* time in *gradient* calculations for problems with large number of parameters (e.g., ratio 3:1); for simple problems, a number of points in gradient and linear search are similar (1:1). Note that the time spent for gradients is definitely not wasted. It gives a big advantage and generally gradient optimizers tend to be more efficient than non-gradient optimizers (like Nelder-Mead).

During optimization, effort for linear search should go down, because the closer to the optimum, the better the quadratic approximation should be; whereas for gradient calculations the effort typically goes up, because at the end it is often better to use double-sided gradient formula, because if  $G$  is not accurate the optimizer may stop too early.

On the other hand, in principle circuit simulators could derive gradients faster than normal simulations by using the adjoint network method for AC simulations or by reusing the values from existing simulations, like in pss analysis.

Another approach to reduce the number of points for gradient calculation is parameter prioritization, e.g., which do not calculate the  $G$  component for parameters on which the user thinks that sensitivity is low anyway and introducing such 2<sup>nd</sup> order parameters *later* into the optimization. However, this approach makes only sense in big optimization problems, which indeed include such 2<sup>nd</sup> order parameters.

## Line Search Analysis

As a significant number of optimization points is needed in line search  $X_{i+1} = X_i + tS$ , the important question is how *accurate* it should be? Most references claim that quasi-Newton and especially BFGS is very robust and already a *moderate* accuracy is good enough. That might not be the case for others, like conjugate gradient methods.

Playing with accuracy parameters shows some impact; of course there is a *trade-off*: Making accuracy quite low may result in bad approximation of inverse Hessian and to more iterations in the *overall* algorithm, which compensates the advantage of less points in the line searches.

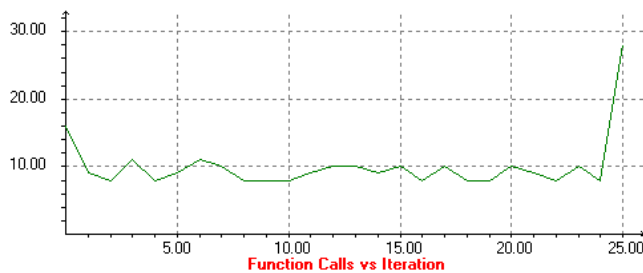
Any gradient optimizer spends the time either in gradient evaluations to determine the search direction  $S$  or in linear search with calculations of  $f$  at  $X_{i+1} = X_i + tS$ . If many used steps  $t$  are too small then time is wasted, if  $t$ 's are large, then maybe also time is wasted too and a further risk is that the circuit may behave badly with too extreme parameters, e.g., an op-amp to be optimized becomes unstable or in a filter a notch might be placed in the passband leading to extreme sensitivities. One good way to avoid extreme parameter is simply limiting the stepsize to  $\pm 50\%$ . Clamping can be done for  $t$ —affecting all parameters—or for each parameter individually.

It has been found that for well-behaved optimization problems the clamping may lead to a small slowdown of the optimization (e.g., 20% more iterations). However, for more nonlinear problems (e.g., due to bad start parameters) there could be a significant speed advantage, e.g. 25% less

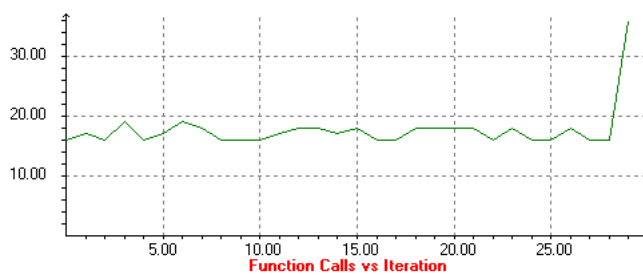
iterations. Beside the number of iterations also the simulations to get the goal function maybe faster if extreme parameter sets are avoided. For some extreme parameter sets it could be that a transient simulation completely fails (e.g., due to "time-step too small") due to circuit instability or extreme time constants, etc.

For other—less-damping—optimizers (like Gauss-Newton or pure Newton) clamping is more important; they take more extreme search directions with large angle of  $S$  to  $-G$ .

Number of calls per iteration for BFGS with single-sided gradient SSG:

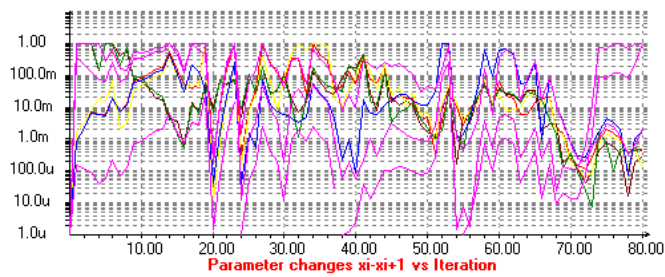
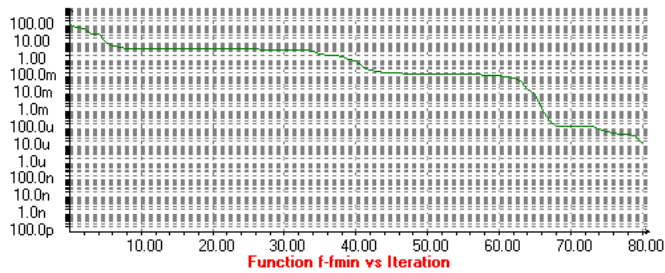
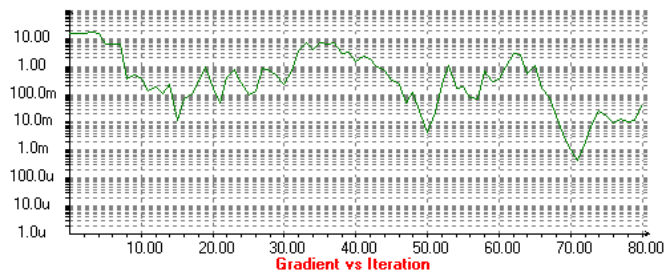


Same with double-sided gradient DSG:

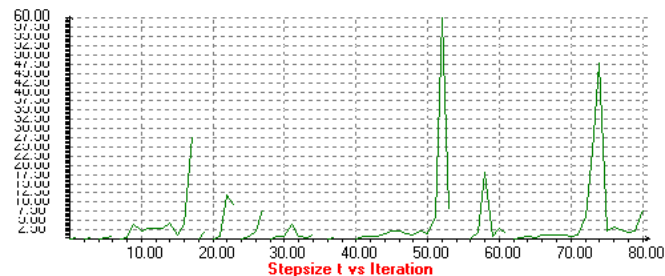
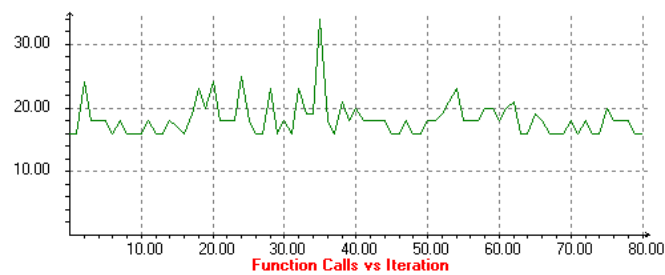


Both histories show similar behavior at the beginning becomes both SSG and DSG are accurate enough, but dsg optimization goes on longer (e.g., finds the optimum more accurately). Also both show many iterations at the end, just before optimizer quits. Having many points in linear search is typical also at the beginning, here finding a good start  $t$  helps too, by checking the individual step  $t \cdot S_i$ . Obviously time can be saved by avoiding too many points in linear search, but on the other hand the optimizer should not stop too early, well before the optimum is reached.

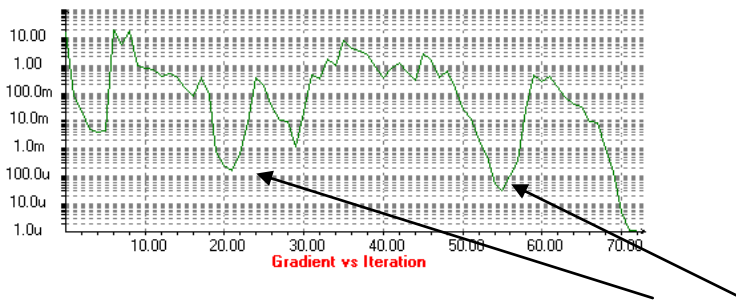
Here is a history example with very bad start parameters: with clamping



Note that the gradient decreases not monotonically at all; also in between there are 2 plateaus of  $f$  where progress seems to be quite small, although some parameters still change quite much. That corresponds also to stepsize and points history.



Without clamping the gradient history (and all others) looks *more extreme*:



Note: A bad optimizer would stop too early, e.g., at iteration 20 (and 55) the gradient  $|G|$  is already small, although the solution is not reached at all.

Another question is whether the clampings may corrupt the calculation of the inverse Hessian. Experiments have shown that this is not really the case.

Last pictures show that at the end of the optimization, the number of points in line search often increases, especially if there are no other reasons to stop the optimization, like reaching the maximum number of iterations or the gradient limit. That high number of points in line search is quite typical and hard to avoid, but costs also some time in more complex optimization tasks. On the other hand, the optimization should not stop too early, just before the optimum is reached. The reason for large number of points in line search is often that the search direction is bad, due to noise in  $G$  or  $H^{-1}$  which may result in need for extreme stepsizes. All-in-all the line search termination depends also on many other more global settings like stepsize  $dx$  used in gradient calculations, or type of gradient approximation (SSG vs. DSG), reset of iteration matrix, etc.

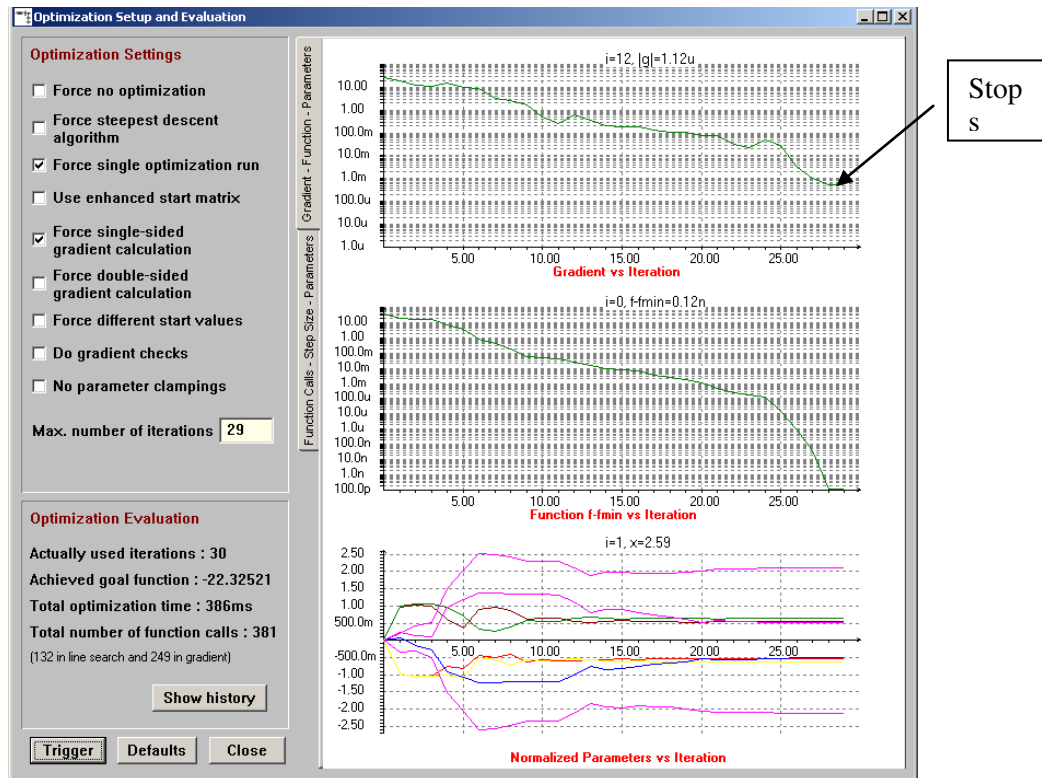
As steepest descent is a very slow optimization method; it is no surprise that the angle between the BFGS or Newton search direction  $S$  can be quite large, like  $85^\circ$  or more. Another interesting monitor is looking at the angle between  $S$  that would come from true-Newton and that from BFGS. The interesting finding is that these two  $S$  have very similar *direction*, although the inverse Hessian approximation by BFGS may look at least not so perfect. Usually the differences in length are much larger (e.g.,  $2x$ ), but there are no problems due to line search algorithm.

## Numerical Noise and Stepsize

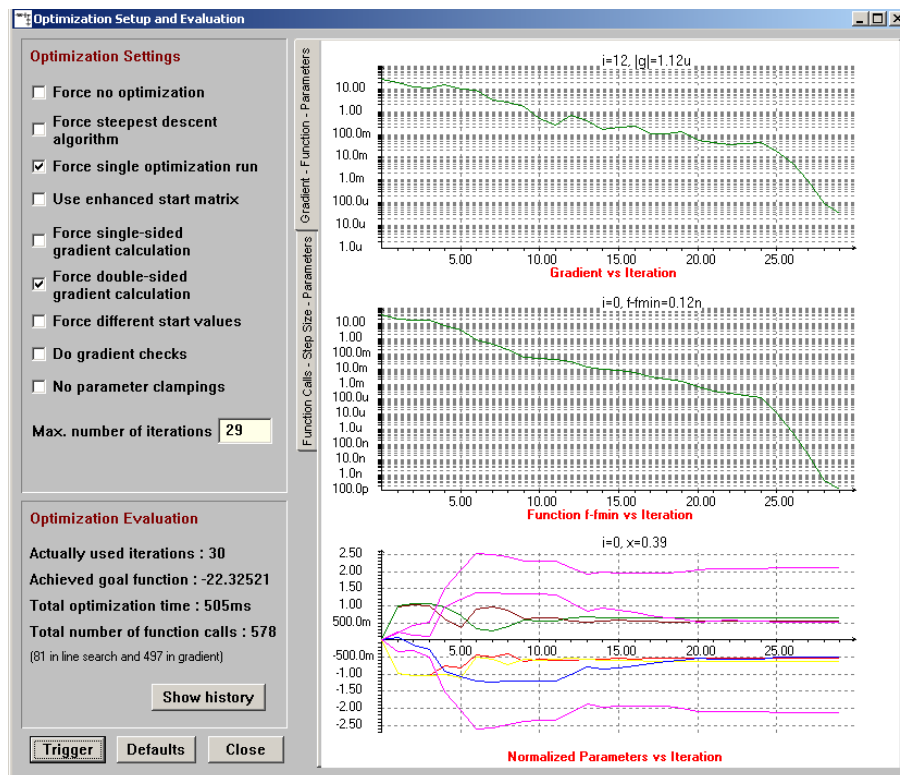
If the goal function evaluation is numerically noisy, e.g. if it comes from a transient analysis with relaxed reltol setting, then the gradient evaluation by finite differences  $dy/dx$  becomes inaccurate. One solution is to increase the gradient stepsize  $dx$  (initially  $dx = 1e-6$ ), but that makes the local truncation error LTE larger (depending on  $d^2f/dx^2$ ), so a compromise is needed (around  $20e-6$  for ANPASS and this optimization task).

Without noise, for  $dx = 20e-6$  the single-sided gradient formula leads to slightly less accurate results on parameters. For DSG much larger  $dx$  can be chosen like  $2E-3$ , still being accurate enough.

Optimization with single-sided gradient approximation:  $dx = 20e-6$



Optimization with double-sided gradient approximation: same  $dx = 20e-6$



With noise the accuracy degrades to app. 0.1% and 0.01% for simple optimization problems and to app. 10% for the complex LC bandpass.

With  $dx = 100e-6$  in the gradient approximation (using SSG) and a noisy simulation we get good performance, but this  $dx$  is quite too large for non-noisy simulations, i.e., optimization stops significantly earlier than with smaller  $dx$  resulting in significantly less accurate, but still highly acceptable parameters.

Without noise,  $dx$  can be even as small as  $0.02e-6$  which is of course related to the very accurate double-precision AC simulation. For goal functions including noisy transient results,  $dx = 200e-6$  or larger is maybe even best.

Generally single-sided gradient formula is accurate enough for *most* optimizations, and of course faster, also as typically ANPASS-BFGS spends more time on gradients than within linear search, so double-sided formula should be used:

- at the end phase to get best accuracy (not only at last interaction)
- for problems with generally small gradients or with local minima
- for problems where highest parameter accuracy is desirable, like model parameter fitting (package problems, extracting transistor parameters, coil model fitting, etc.)
- for problems where numerical noise is large leading to need for large  $dx$ , i.e. unacceptable large LTE for single-sided gradients.
- If PDK has built-in snap-to-grid, that would need quite large  $dx$ , so that single-sided gradient is not accurate enough due to LTE.

So all-in-all it makes sense to let the optimizer dynamically switch between single-sided and double-side gradient.

As gradient is large at the beginning and single-sided gradient approximation (SSG) is faster, it should be used first. However, the final accuracy is often *significantly better* with double-sided gradients (DSG). So the question arises where is the best point for switching between SSG and DSG to get both best efficiency *and* accuracy. If switching is too late, then there is a big risk that inverse Hessian matrix  $iH$  is often already numerically distorted due to inaccuracies from SSG. So generally a good switching point is that the last approximately. 30% BFGS iterations are done with DSG. The major problem is how to know in advance, when this happens? Multiple criteria are used to detect that, depending on current gradient values.

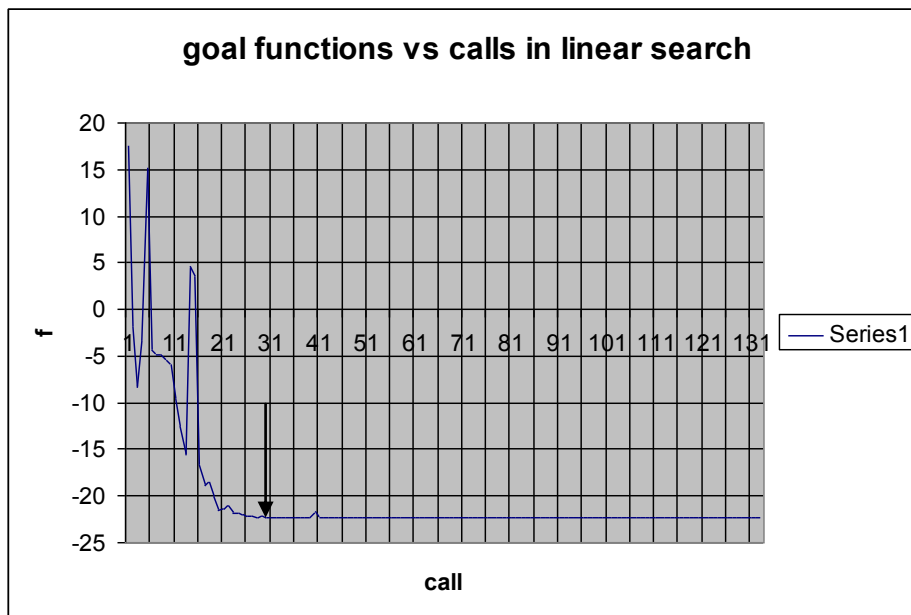
Note that it is sometimes dangerous to start with SSG and to wait checking on switching to DSG, because in some situations the start point might be already difficult, which requires DSG already from the beginning. So, if needed, the switching to DSG should be possible already from the beginning.

Currently ANPASS uses app 50% SSG and 50% DSG, which is a good compromise between speed and accuracy.

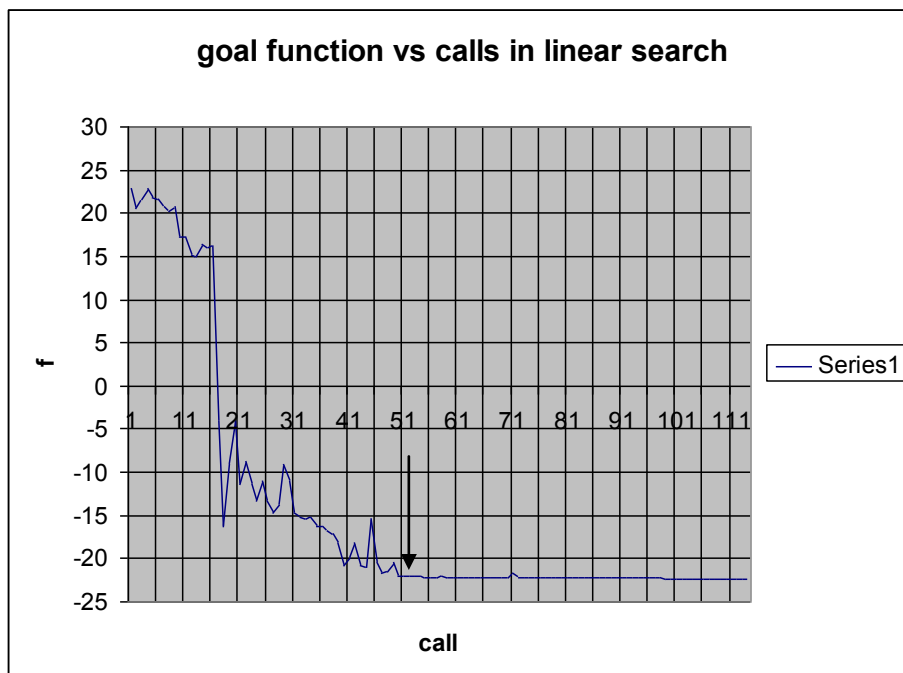
## Goal Function Definition, Different Starting Points and Local Minima

Indeed, all optimizers are quite sensitive on goal function definition, but a good optimizer should still work, also on  $f$  using max-norm  $L_\infty$ , although some performance loss (i.e., larger number of calls/points) must be usually accepted. ANPASS typically requires  $2x$  more points and stops sooner, i.e. at larger final gradient and therefore less accurate parameters. These problems are also found by inspection of Hessian matrix; it often has negative eigenvalues and does not really correlate well with matrix used in BFGS in opposite to using average in  $f$  (instead of max).

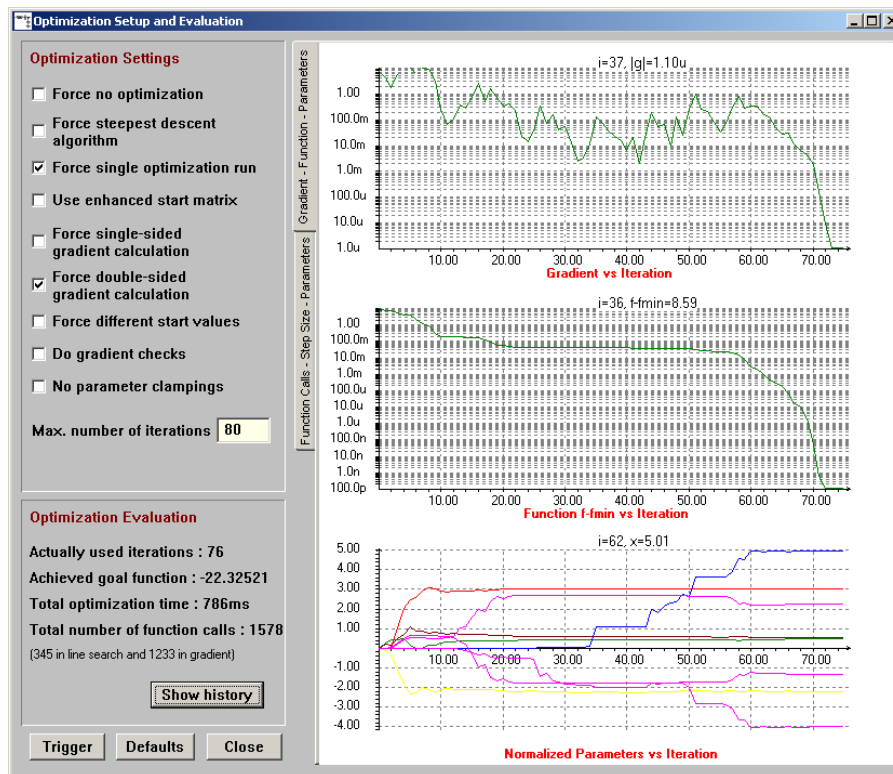
Goal function history using *average*: gradient calculations excluded, because their  $\Delta f$  is very small



Goal function history using *max-norm*: takes more time



Setting all L's to 1 uH and all C's to 1 pF—i.e. parameters are offset by up to 150×—leads to that optimization history:



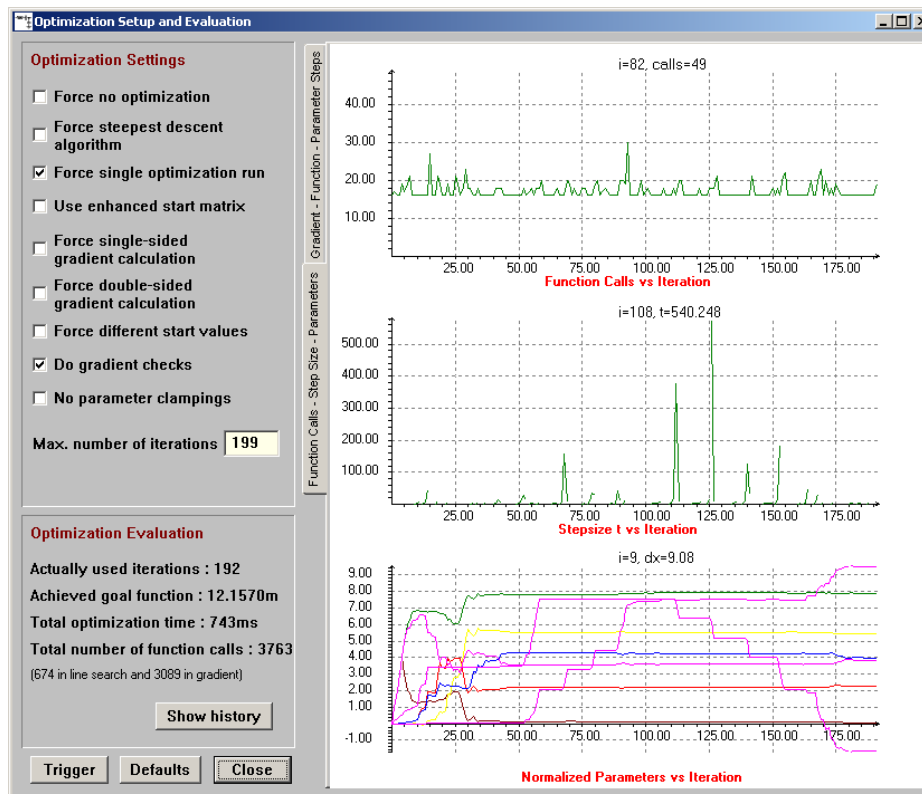
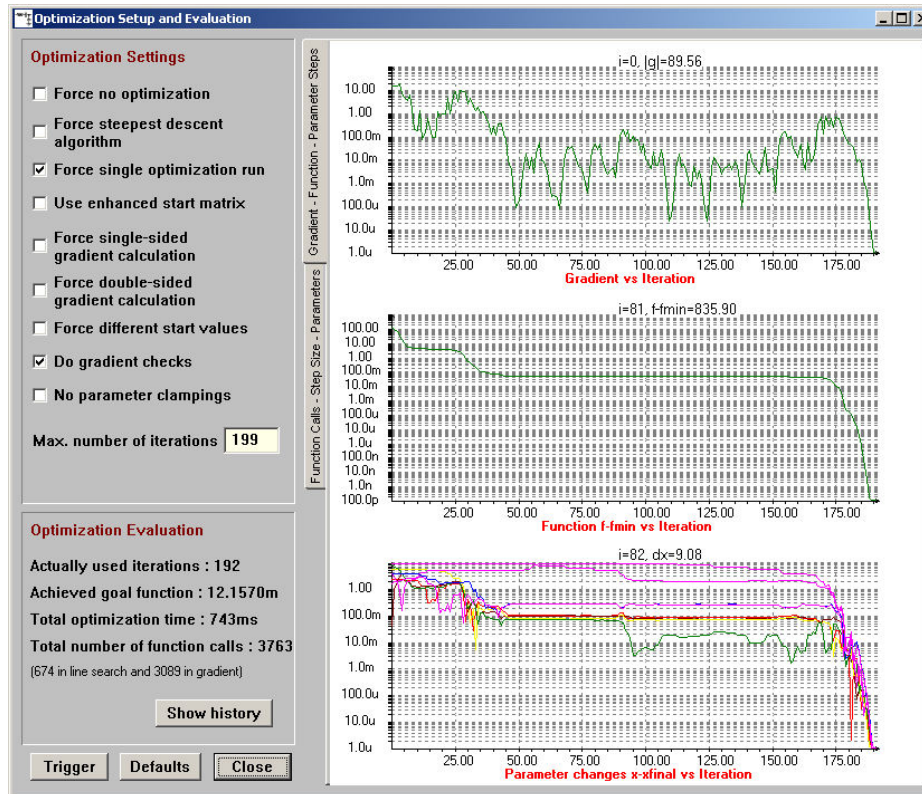
Remember: For pure quadratic problems BFGS needs  $N$  iterations, no matter of the starting point.

Here for a general nonlinear optimization the number of iterations depends on the start point (need for 76 iterations instead of 30).

This shows the robustness of BFGS and even more extreme start parameters (like all L's 1 nH) are no problem in this optimization task! Note that  $f$  decreases *monotonically*, but  $|G|$  *not*. If max-norm is used instead of average, then optimization still works, but slower and less accurate. Here a simple restart helps best, also parameter stepsize limiting ( $dx_i \leq 1$ ) helps to speed-up (1.5×).

Also other kind of different starting point sets have been tested: like all parameters 10× bigger than the final solution, or all L at 0.1× and all C at 10×. Luckily these parameter shifts are leess hard to optimize. BFGS usually needs few iterations only.

Here are the results of a very hard optimization run: Setting all L's to 1 nH (not 1 uH) and all C's to 1pF,  $R_L = 5k$ ,  $R_G = 50\Omega$ , leads to that optimization history with need for 200 iterations:



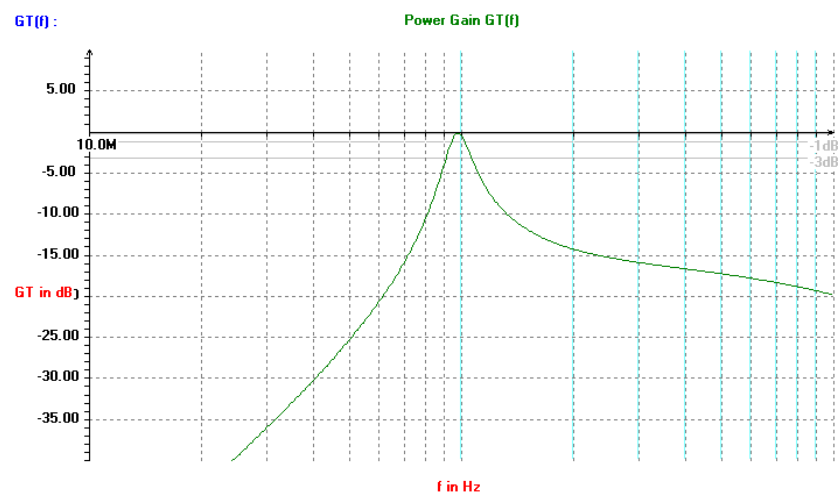
Note: At multiple times  $|G|$  becomes very small (like at 53 iterations). Also look up that even at last iterations there are still significant progresses ( $10\times$  changes!) in finding the best parameter values. After 53 iterations the gain vs.  $f$  curve is already quite fine, but of course not yet the fully optimized Chebyshev low-ripple characteristic. With SSG-only BFGS would stop here, but would work with 2 restarts.

For even more extreme start parameters BFGS may fail and can stop at a local minimum, i.e., it could happen that one L and C is ‘optimized’ to a near-0 value—an example of parameter redundancy which would degrade the bandpass filter order. A good way is then to check the filter behavior in a Smith chart. You will note that then some elements have no real impact on filter behavior (e.g., a big series-C acts like a wire). In this case a designer would need to decide on whether just skipping that elements in the design (and accepting non-optimum performance) or to retune the bad elements and to rerun the optimizer. That is possible with ANPASS too. Changing L4 and C4 manually (to quite some arbitrary value) and restarting BFGS leads (usually) to the desired global optimum.

If we want to get rid of such situations in a global optimizer, then such things are easy to implement, but on the other hand it can be also ‘dangerous’ to shift parameters a lot because real nonlinear circuits may run into problems (bad DC operating point, oscillations, etc.).

For filter optimization, a simple reason for fail in case of bad starting points is also that for bad element values the AC simulation setup may not even cover the whole bandpass behavior but maybe on the beginning of it, where the bandpass just looks like a highpass filter! In more complex transistor circuits, often bad operating regions for important transistors “kill” the optimization progress.

Also this optimization task has local minima, which sometimes are found, if max-norm is used (or if the starting point is extremely bad).



Parameters:

|    |         |   |    |        |   |
|----|---------|---|----|--------|---|
| L1 | 277.56p | H | C1 | 6.13p  | F |
| L2 | 403.11n | H | C2 | 6.66p  | F |
| L3 | 810.98n | H | C3 | 3.47p  | F |
| L4 | 1.02n   | H | C4 | 46.57p | F |

It seems that in such cases, numerical noise can help to get out of the local minima.