LISO - Program for **Li**near **S**imulation and **O**ptimization of analog electronic circuits – **Version 1.61**

G. Heinzel, MPQ Garching

March 23, 1999

Contents

1 Introduction 4

2 Availability 4

3 Features and limitations 5

4 Invoking LISO 7

4.1 Options 7

5 The input file 8

5.1 General remarks 8

5.2 Instruction summary 9

5.3 Example 12

6 Circuit description 13

6.1 Overview 13

6.2 Nodes 14

6.3 Passive components 15

6.4 Mutual inductances 15

6.5 Op-amps 15

6.6 Input definition 16

6.7 Input factor 16

7 Circuit mode: requesting output 17
7.1 Transfer functions . . . 17
7.1.1 Treatment of the input . . . 17
7.1.2 Specifying the output format . . . 18
7.1.3 Voltage transfer functions . . . 18
7.1.4 Current transfer functions . . . 19
7.1.5 Maximum permissible input signal . . . 19
7.1.6 Input impedance . . . 20
7.1.7 Op-amp differential input voltages . . . 20
7.2 Op-amp stability . . . 21
7.3 Noise calculations . . . 21

8 Root mode 23
8.1 Transfer function definition . . . 24
8.1.1 Poles and zeroes . . . 24
8.1.2 Overall factor . . . 25
8.1.3 Frequency scaling . . . 25
8.1.4 Delay time . . . 25
8.2 Requesting output of the transfer function . . . 25

9 Fitting 25
9.1 Introduction . . . 26
9.2 Function to be fitted . . . 26
9.3 Data file . . . 27
9.3.1 Weighting . . . 28
9.4 Parameters . . . 29
9.4.1 Dependent parameters . . . 30
9.5 Additional fitting constraints for optimizing the dynamic range . . . 31
9.5.1 Input range constraint . . . 32
9.5.2 Noise minimization . . . 33
9.6 Rewriting the input file with best-fit parameters . . . 33
9.7 Output of the fitting procedure . . . 34
9.8 Hints for fitting . . . 36

10 Plot options 39
10.1 Defining the frequency range . . . 39
10.2 GNUPLOT terminals . . . 39

11 Initialization file “fil.ini” **40**

12 Op-amp library **42**
12.1 Open-loop voltage gain . . . 42
12.2 Op-amp noise . . . 44
12.3 Op-amp limits . . . 45

13 Circuit simulation algorithm **45**
13.1 Transfer functions . . . 45
13.2 Noise calculations . . . 48
13.2.1 Treatment of the input . . . 48
13.2.2 Resistor noise . . . 48
13.2.3 Op-amp voltage noise . . . 49
13.2.4 Op-amp current noise . . . 49
13.3 Op-amp stability calculations . . . 50

14 Fitting algorithms **51**
14.1 Extensions to the Nelder-Mead Simplex algorithm . . . 51
14.1.1 Computation of $n + 1$ directions equally distributed in R^n . . . 51
14.1.2 Derivation . . . 52
14.1.3 Example . . . 53
14.1.4 Initialization and termination . . . 53
14.2 Extensions to the Levenberg-Marquardt algorithm . . . 53
14.2.1 Numerical derivatives . . . 53
14.2.2 Application to complex data . . . 54
14.2.3 λ search . . . 55
14.3 Overall minimization strategy . . . 56
14.4 Implementation of parameter limits . . . 56
14.5 Correlation matrix and confidence limits . . . 58

15 Implementation notes **59**
15.1 Internal scaling . . . 59
15.2 Sparse matrices . . . 59
15.3 Efficient noise computation using the transposed matrix . . . 60

16 Application examples **61**
16.1 Root mode simulation . . . 61
16.2 Root mode fit . . . 62

1 Introduction

The computer simulation of electronic circuits is a well-established discipline. The widely known program SPICE, e.g., is very powerful at simulating things such as MOS transistors, transient behaviour, integrated circuits, digital logic devices, etc. For many of the circuits that we need, however, the emphasis is on other aspects:

- The frequency response of preamplifiers and active filters built with op-amps in the presence of non-ideal op-amp characteristics (finite gain and strange phase shifts around the unity gain frequency).
- The noise behaviour of such circuits.
- The stability of each op-amp in a complex circuit.
- The desire to optimize some components of such circuits for a given purpose, such as maximum dynamic range.

The frequency response of several interesting circuits had been found analytically by the author using MATHEMATICA, assuming ideal op-amps. The attempt to extend these analytical models to real op-amps showed, however, that the analytical solutions very quickly become too complex to be useful. In particular the noise behaviour gets very complicated to compute. SPICE models provided by op-amp manufacturers do usually not correctly predict the noise behaviour of the circuits. Therefore a program was written that computes these aspects of a given circuit numerically.

Using the simulation as base, the additional **fitting** function is a powerful tool to analyze, design and optimize circuits.

2 Availability

The program and this manual were written by:

Gerhard Heinzel
Max-Planck-Institut für Quantenoptik
Hans–Kopfermann–Straße 1
D–85748 Garching
e-mail: `ghh@mpq.mpg.de`

mainly for use by members of GEO 600 and cooperating projects. Both are

Copyright © G. Heinzel 1997, 1998, 1999.

Various portions of software from other authors were used. These have copyrights of their own, which allow free non–commercial use under certain conditions, such as complete unmodified distribution with all sources. Since at present this is not yet the case, I ask everyone:

- Feel free to use LISO for any noncommercial application.

- Of course you may also give it to interested colleagues and friends.
- Please to *not* distribute LISO to the general public, such as by placing it on your own website.
- For any wider distribution or non-scientific application, please contact the author first.
- The author is interested to learn about the applications, usefulness, problems, wishes, bugs etc. of LISO.
- Of course no guarantee of any kind is given or implied, such as for the correctness of any algorithm, model or result. Any application of LISO is at the full risk of the user.

LISO is available by `ftp` from:
`ftp.rzg.mpg.de` in the directory
`/pub/grav/ghh/liso`.

3 Features and limitations

The input to LISO is a fixed circuit consisting of passive components (any combination of R, C, L, transformers) and op-amps. It can have a voltage input (grounded or floating) or a current input (such as for photodiode preamplifiers). The program then computes and plots:

- The frequency response from the input to any component or node (in particular, the voltage at the output or the current through any component).
- The maximum permissible input signal, taking into account op-amp output voltage, output current and slew rate.
- The stability of each op-amp in the circuit. Since op-amps are the only elements with gain, this is equivalent to computing the stability of each closed loop in the circuit and the stability of the circuit as a whole.
- The voltage noise at any node of the circuit (in particular the output), taking into account Johnson noise of resistors and voltage noise and current noise of op-amps. It can separate the individual contributions of these noise sources and identify the dominating noise source for any frequency range.
- The spectral density of the noise current in each component, taking into account the same noise sources as in the above item.

Some features of the program include:

- The program uses a user-expandable library of op-amp models with their main characteristics. For each op-amp in the circuit, the library parameters can be "overridden" by individual parameters.

- A very important function of the program is its ability to **fit** the model to given data (either measured or ideal desired data) by varying specified components or parameters.
- All computations are done complex. In particular, data measured with phase can be properly fitted.
- Some effort has been made to incorporate state-of-the-art algorithms for the simulation and fitting parts, i.e. the author believes that the results are accurate for reasonable inputs and they are delivered fast.
- In a separate mode of operation of the program, frequency responses can be computed using the poles and zeroes of the transfer function as input instead of a circuit description.
- Extensions to the basic fitting algorithm allow the user to find circuits with a user-specified transfer function under additional constraints such as a minimal allowed output swing and minimum noise.
- Two new fitting procedures *Direct Search Simulated Annealing* and *Controlled Random Search* often converge to solutions even when no reasonable starting value are given.
- The combination of the last two items allows the user to almost automatically find circuits with a specified transfer function and the maximum possible dynamic range. Many of these solutions are almost impossible to find by other means.
- The results are plotted via GNUPLOT, which supports many different output 'devices' (including Postscript) and allows the user to change the appearance of the plot in an easy and flexible way.

The program has at present several limitations. Some of them (marked with an asterisk *) could possibly be overcome if necessary, others not.

- The circuit topology is fixed and must be known and entered by the user.
- All computations are done linearly in the frequency domain. In particular, the DC operating point is not computed. No time-domain analyses are performed. No non-linear components such as discrete semiconductors can be simulated.
- Only voltage-feedback op-amps can be simulated. *
- There is no graphical user interface. Input and output are done via ASCII files in a batch-mode like operation. Plots are produced by calling GNUPLOT. Without GNUPLOT, the only output is an ASCII data file, which may be plotted by other software.
- At present the program has mainly been tested under various versions of UNIX (mainly LINUX, also SUN and IBM) and MS-DOS. There should, however, be no big problem in porting it to any reasonable operating system. It is entirely written in C and the preferred compiler is the GNU C-compiler.

- Fitting is only possible for the "transfer functions", not for noise. *
- There is only a very simple electrical rule check (ERC) which will not detect many kinds of errors in the circuit (e.g. outputs connected to GND etc.) *
- Not all useful transfer functions can be entered by the present `pole`/`zero` syntax.*

4 Invoking LISO

LISO is started from the command line with one, two or three filename arguments. In order to run, LISO needs three filenames: the name of the input file, which usually has the extension `.fil`, the output file which usually has the extension `.out`, and the GNUPLOT batch command file, which usually has the extension `.gnu`.

The easiest way is to give only one argument, the *basename* for all files involved. For example, calling

`liso` [*options*] `test`

is equivalent to

`liso` [*options*] `test.fil test.out test.gnu`

If you want to use different filenames, you can call LISO with three arguments:

`liso` [*options*] *input-file output-file gnuplot-file*

If you give only two file names on the command line, LISO tries to extract the basename from the name of the input file and appends `.gnu` to it to construct the gnuplot file name. E.g. calling

`liso` [*options*] `test.fil output-file`

is equivalent to

`liso` [*options*] `test.fil output-file test.gnu`

LISO will first read and interpret the input file. If there are no errors, the computations specified in the input file will be performed. Their results are written to the output file. Some intermediate status information will simultaneously be written to the console (`stdout`). At the end of the output file, LISO writes a copy of the input, as it was interpreted. This is not only useful for documentation purposes, but also for debugging, if LISO produces unexpected results. Finally LISO writes the GNUPLOT command file and calls GNUPLOT to display the specified plots.

4.1 Options

At present there are only two options:

`-m` requests that only the Marquardt algorithm should be used for fitting, i.e. the Simplex algorithm is disabled.

`-s` requests that only the Simplex algorithm should be used for fitting, i.e. the Marquardt algorithm is disabled.

These are sometimes useful in special fitting situations (see Section 9.8 on page 36). In general, however, the combination of both algorithms (no option given) yields best results.

5 The input file

5.1 General remarks

The input file is a plain-text ASCII file. Each line is interpreted as one instruction, identified by a keyword. In the following we speak of *instructions* (e.g. a `freq` instruction) and mean by that a line in the input file that begins with that keyword. Most of the input file is interpreted case-insensitively, i.e. `opamp1`, `OpAmp1` and `OPAMP1` are all the same thing. The only exceptions are the abbreviations listed below. Empty lines are ignored. A '`#`' indicates that the rest of the line is a comment and will be ignored.

Numbers can be entered in the following formats: `2200`, `2.2e3` `.22e4` or `2200.0`. Additionally, the following abbreviations are recognized (and used in outputs):

abbreviation	meaning	value
G	giga	10^9
M	mega	10^6
k	kilo	10^3
m	milli	10^{-3}
u	micro	10^{-6}
n	nano	10^{-9}
p	pico	10^{-12}
f	femto	10^{-15}

These abbreviations *are* case-sensitive. As an example, the above number could also be written as `2.2k` or even `0.0022M`.

The *units* for all numbers in LISO, in the input as well as the output, are the SI-units. In particular the following units are used:

Ω for resistors (written `Ohm` in the output),
F for capacitors,
H for inductors,
V for voltages,
A for currents,
Hz for frequencies,
$V/\sqrt{Hz}$ for voltage noise spectral densities (written `V/sqrt(Hz)` in the output) and
$A/\sqrt{Hz}$ for current noise spectral densities (written `A/sqrt(Hz)` in the output).

In the input file, these units are *not* entered. For example, a capacitance of 33 pF could be entered as `33p` or `3.3e-11`. Note that all frequencies in LISO are physical frequencies f, and *not* angular frequencies $\omega = 2\pi f$.

5.2 Instruction summary

The program has two main modes of operation. They are called *circuit mode* and *root mode*. In circuit mode, the input consists of a circuit description (comparable to “nodelists” of other programs), and the output are transfer functions, noise spectra, etc. of that circuit. In root mode, the input consists of a list of poles and zeroes of the transfer function, and the output is just the transfer function itself (as a function of frequency). In any single run of the program, only one of the two modes can be used. By running the program several times with different input files, however, these two modes can be combined in various useful ways.

To describe the required syntax of the input file, we introduce the following conventions:

- Words written in `typewriter` style must be entered literally.
- Words written *slanted* such as *component-name* must be replaced by the appropriate input.
- Words included in brackets [ ] are optional.
- Words separated by vertical lines | represent alternatives, i.e. exactly one of the alternatives must be entered.
- Three dots ... indicate that a repetition is allowed.

The instructions available in circuit mode are:

`r` *name value node1 node2*
: defines a resistor (see section 6.3 on page 15).

`c` *name value node1 node2*
: defines a capacitor (see section 6.3 on page 15).

`l` *name value node1 node2*
: defines an inductor (see section 6.3 on page 15).

`m` *name value inductor1 inductor2*
: defines a mutual inductance (such as in transformers, see section 6.4 on page 15).

`op` *name type node+ node− nodeout* [*parameter*=*value* ...]
: defines an op-amp (see section 6.5 on page 15).

`uinput` *input-node [input-node2] source-impedance*
: defines a voltage input to the circuit (see section 6.6 on page 16).

`iinput` *input-node source-impedance*
: defines a current input to the circuit (see section 6.6 on page 16).

`factor` *factor*
: defines a factor which multiplies the input (see section 6.7 on page 16).

The following instructions in circuit mode are used to request output:

`uoutput all`[*coordinates*]
: requests the voltage at all nodes to be plotted (see section 7.1.3 on page 18).

`uoutput allop`[*coordinates*]
: requests the voltage at all op-amp outputs to be plotted (see section 7.1.3 on page 18).

`uoutput` *node*[*coordinates*] [*node*[*coordinates*]] ...
: requests the voltage at one or several nodes to be plotted (see section 7.1.3 on page 18).

`ioutput all`[*coordinates*]
: requests the current through all components to be plotted (see section 7.1.4 on page 19).

`ioutput allop`[*coordinates*]
: requests the current through all op-amps to be plotted (see section 7.1.4 on page 19).

`ioutput` *component*[*coordinates*] [*component*[*coordinates*]] ...
: requests the current through one or several components to be plotted (see section 7.1.4 on page 19).

`maxinput`
: requests the maximum permissible input to be plotted (see section 7.1.5 on page 19).

`zin`[*coordinates*]
: requests the input impedance to be plotted (see section 7.1.6 on page 20).

`opdiff all`[*coordinates*]
: requests the differential voltage between the two inputs of all op-amps to be plotted (see section 7.1.7 on page 20).

`opdiff` *op-name*[*coordinates*] [*op-name*[*coordinates*]] ...
: requests the differential voltage between the two inputs of one or more op-amps to be plotted (see section 7.1.7 on page 20).

`opstab all`[*coordinates*]
: requests the "stability function" of all op-amps to be plotted (see section 7.2 on page 21).

`opstab` *op-name*[*coordinates*] [*op-name*[*coordinates*]
: requests the "stability function" of one or more op-amps to be plotted (see section 7.2 on page 21).

`:db|abs|re|im|deg|deg+|deg-[:db|abs|re|im|deg|deg+|deg-]` ...
: is an option (called *coordinates*) for the above instructions that specifies the output format (see section 7.1.2 on page 18).

`noise` *node* `sum|all|allr|allop|`*noise-source* [`sum|all|allr|allop|`*noise-source*] ...
: requests the noise at a certain node of the circuit to be plotted (see section 7.3 on page 21).

`noisy` `all`|`allr`|`allop`|*noise-source* [`sum`|`all`|`allr`|`allop`|*noise-source*] ...
: defines which noise sources should contribute to the plotted total noise (see section 4 on page 22).

`:u|+|-` [`u|+|-`]
: is a suffix for an op-amp name in the above two instructions that switches on/off individual noise contributions of an op-amp (see section 4 on page 22).

The instructions available in root mode are:

`pole` *frequency* [*Q*] defines a pole of the transfer function (see section 8.1.1 on page 24).

`zero` *frequency* [*Q*] defines a zero of the transfer function (see section 8.1.1 on page 24).

`factor` *factor*
: defines an overall factor of the transfer function (see section 8.1.2 on page 25).

`ffactor` *scale-factor*
: defines a frequency scaling factor of the transfer function (see section 8.1.3 on page 25).

`delay` *delay-time*
: defines a delay time in the transfer function (see section 8.1.4 on page 25).

`tfoutput` *coordinates*
: requests the transfer function to be plotted (see section 8.2 on page 25).

Two instructions are common to both modes:

`freq` `lin`|`log` *startfreq stopfreq steps*
: defines the frequency range for all plots (see section 10.1 on page 39).

`gnuterm` *terminal-name [file-name]* selects an output terminal for GNUPLOT (see section 10.2 on page 39).

The following instructions are available in both modes, if a fit is to be performed:

`param` *parameter-name lower-limit upper-limit*
: defines an independent parameter to be fitted (see section 9.4 on page 29).

`sparam` *parameter-name*
: defines a dependent parameter that is equal to the last independent parameter (see section 9.4.1 on page 30).

`pparam` *parameter-name* [*factor*]
: defines a dependent parameter that is proportional to the last independent parameter (see section 9.4.1 on page 30).

`rparam` *parameter-name* [*factor*]
: defines a dependent parameter that is inversely proportional to the last independent parameter (see section 9.4.1 on page 30).

`fit` *file-name* `reim`|`dbdeg`|`absdeg`|`db`|`abs` `abs`|`rel`|`semi`|`sdev` [`plotfirst`]
: describes the file holding the data to be fitted and requests the fit (see section 9.3 on page 27).

`rewrite` [*file-name* | `same`]
: causes a copy of the input file to be written, where all parameters that were fitted are replaced by their best-fit results (see section 9.6 on page 33).

`mininput` *value min-freq max-freq*
: (valid only in circuit mode) puts an additional constraint on the fit: the permissible input signal must be bigger than *value* for all frequencies between *min-freq* and *max-freq* (see section 9.5.1 on page 32).

`minnoise` *node min-freq max-freq aim*
: (valid only in circuit mode) puts an additional constraint on the fit: the rms noise at node *node* should be below *aim* in the frequency band between *min-freq* and *max-freq* (see section 9.5.2 on page 33).

5.3 Example

An example of a complete input file is given in Figure 1 on the next page. The meaning of each instruction will be explained in detail in the following sections.

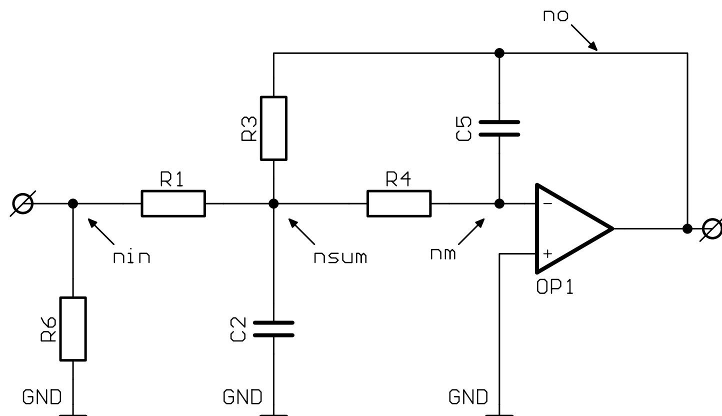
LISO will produce the following output from that input file:

```
LISO analog circuit simulation 1.32  G. Heinzel MPQ 13.4.98 (ghh@mpq.mpg.de)
  Input file x.fil, Output file x.out, Gnuplot file x.gnu
Mon Apr 13 21:39:23 1998

nfunc  Stat       ssq        r1        r4
    1 Start     29.76       100      42.2
    4  Splx      6.87     141.4     51.54
    6  Splx    0.4615     199.9     28.29
   16  Splx    0.2844     212.8      32.1
   22  Splx    0.2119     212.7     30.05
   33  Sout     0.209     213.2     29.66
   43  Marq    0.2086     213.9     29.47
   51  Marq    0.2086     213.9     29.48
   51  Mout    0.2086     213.9     29.48
   54  Splx    0.3833     213.9     26.11
   56  Splx    0.2421     219.6     29.93
   64  Splx    0.2168     213.9     28.71
   78  Sout    0.2087     213.8     29.41

Correlation matrix
            r1      r4
     r1       1
     r4  -0.533       1

Best parameter estimates:
r1 =  213.91856 +- 3.365 (1.57%)
r4 =  29.47639 +- 840.4m (2.85%)
```




```
# circuit definition
r r1 100 nin nsum
r r3 1.075k no nsum
r r4 42.2 nsum nm
r r6 65 nin gnd
c c2 4.7n nsum gnd
c c5 122p no nm
op op1 ad797 gnd nm no pole0=8e6
uinput nin 50

# computing instructions
freq log 10k 10M 400
uoutput no:db:deg

# fitting instructions
param r1 10 10k
param r4 10 10k
fit soll reim rel
```

Figure 1: Sample input file describing the low-pass filter shown above (file `x.fil`).

6 Circuit description

6.1 Overview

In circuit mode, the input file consists of three kinds of instructions:

Circuit definition The circuit is defined by one or more `r`, `c`, `l`, `m` or `op` instructions, and exactly one input instruction (either `uinput` or `iinput`). They define the topology of the circuit and the component values. These may come in any order, but must be completed before any computing instructions or fitting instructions appear. These instructions are described in this section.

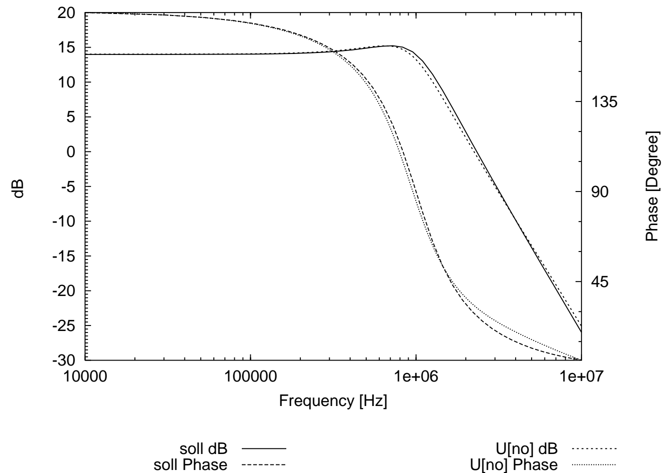


Figure 2: Output from the example in Figure 1 on the preceding page.

Computing instructions These intructions define the type of computation and plot to be done. There must be exactly one `freq` instruction, which describes the frequency range of the plot (unless a fit is done and the frequency range is taken from the data file). There must be one or more `uoutput`, `ioutput`, `maxinput`, `opdiff`, `zin`, `opstab`, `noisy` or `noise` instructions. Some, but not all combinations are allowed (see Section 7 on page 17).

Fitting instructions These instructions are optional and only used if an optimization is desired. If they are absent, the program will only do a simulation of the given circuit. For an optimization, there must be one or more `param` instructions optionally followed by `sparam`, `pparam` or `rparam` instructions. These define the parameters to be varied in the optimization. There must also be exactly one `fit` instruction which describes the input file for the fitting procedure, e.g. the measured or desired frequency response to be approximated and its file format. These instructions are described in Section 9.

6.2 Nodes

An essential concept in the circuit description is the *node*, which is defined as the common connection between components. A node is assumed to be an ideal wire, enforcing the same voltage at all points where it is connected. Nodes must be given names by the user. The names may be alphanumeric, e.g. `n1`, `n_sum_17`, `op3_output`, etc. They are treated case-independently, i.e. `n_sum` and `N_SUM` are the same node. I have adopted the convention to begin all node names with `n`; this is, however, left to the user's taste. Node names may be at most 14 characters long. Nodes cannot to be defined explicitly; the program automatically creates a list of all nodes when it parses the input file. The circuit ground is treated in a special way and must always be entered

as gnd (or GND). All voltages are referred to that ground.

6.3 Passive components

For each passive component in the circuit, there must be exactly one instruction in the input file beginning with either r, c or l for resistors, capacitors and inductors, respectively. Each such instruction needs exactly 5 entries in the following form:

r|l|c *name value node1 node2*

The *name* is a character string of up to 14 characters, which is used in later in- and outputs to identify the component. The *value* must be given in the appropriate SI-unit (Ohm, Farad or Henry), but without explicitly entering the unit. The abbreviations M, k, m, etc. for powers of ten as listed in Section 5.1 on page 8 may be used. Finally, the two nodes *node1* and *node2* between which the component is connected, must be given.

6.4 Mutual inductances

Transformers and similar devices are described by mutual inductances. All windings of the transformer must be defined as individual coils. Then a coupling factor (between zero for independent coils and unity for a perfect transformer) is introduced, which couples the various inductances to each other. The syntax of the instruction is:

m *name value inductor1 inductor2*

The *name* is a character string of up to 14 characters, which is used in later in- and outputs to identify the component. The *value* is the coupling factor, which must be between zero and unity. The two inductors to be coupled together are identified by their names *inductor1* and *inductor2*. Here is an example of a transformer with a winding ratio of 1:10 and a primary inductance of 10 μH :

```
l lprim 10u np1 np2
l lsec 1000u ns1 ns2
m m1 0.95 lprim lsec
```

Note that impedances (in particular inductances) transform with the square of the winding ratio. Note also that it is not allowed to create parts of the circuit that are completely floating. The simulation algorithm will fail in these cases. At least one point of the secondary circuit must be connected to ground, otherwise the simluation algorithm will fail (because all voltages are referenced to ground).

6.5 Op-amps

For each op-amp in the circuit, there must be exactly one instruction in the input file with the following format:

op *name type node+ node− nodeout* [*characteristic=value* . . .]

The *name* is a character string of up to 14 characters, which is used in later in- and outputs to identify the op-amp. The *type* is another character string, such as op27,

which must be defined in the op-amp library (see Section 12 on page 42). The three node names define how the op-amp is connected to the rest of the circuit, their order is non-inverting input, inverting input and output. Finally there may be zero or more "override" instructions. These have the form *characteristic=value*, where *characteristic* is one of `a0, gbw, un, uc, in, ic, delay, pole0 ... pole4, zero0 ... zero4, umax, imax` or `sr`. The meaning of these characteristics is explained in Section 12. Their purpose is to allow individual op-amps to have different characteristics than stored in the library. There must be an equal sign '`=`', but no space between the name of the characteristic and the value. An example is shown in the sample input file (Figure 1 on page 13).

6.6 Input definition

The circuit must have exactly one input, which is used for the transfer function calculations. The input can be either a voltage or a current. It is defined by exactly one instruction of the form

`iinput` *input-node source-impedance*
`uinput` *input-node source-impedance*
`uinput` *input-node input-node2 source-impedance*

The *input-node* defines to which node of the circuit the input is applied. The last form (`uinput` with two nodes given) is used to define a *floating* voltage input, which is connected between the two given nodes[1]. For the first two forms, the input is referred to ground.

The *source-impedance* is assumed to be a real impedance between the input nodes (between the input node and GND, in the normal case of a non-floating input). A complex source impedance can be simulated by explicitly connecting the appropriate passive components between the input nodes. Note that for the computation of transfer functions, the *source-impedance* is ignored, whereas for the computation of noise and op-amp stability functions, the distinction between `uinput` and `iinput` is ignored (see sections 7.1.1 on the next page, 7.2 on page 21 and 4 on page 22).

6.7 Input factor

Normally the input to the circuit is taken to be 1 V or 1 A at the frequency of interest. In some special applications, however, it is useful to assume a different input signal. One example is when measured data are to be fitted to a model, and the measured data contain some factor caused by the measurement process[2]. The instruction has the following simple format:

[1] This was used to compute the output impedance of a current source. The normal control input was grounded, and a floating voltage source connected in series with the load. The `zin` instruction was then used to compute the unknown impedance.

[2] One example is the measurement of an unknown complex impedance with a RF network analyzer with 50 Ω ports. For best results the 50 Ω source impedance must be included in the circuit model. However, the analyzer is usually calibrated such that its transfer function output is unity for a short between source and receiver. Using a circuit model that includes the 50 Ω source and receiver impedance and a unity source voltage will however produce a transfer function of 1/2 for this short. A model for

`factor` *factor*

The *factor* can be any positive or negative number. The circuit input is multiplied by that factor. It multiplies voltage outputs (`uoutput`), current outputs (`ioutput`), noise outputs (`noise`) and op-amp differential input voltages (`opdiff`). It does not affect the computation of the circuit input impedance (`zin`) or op-amp stability (`opstab`).

The input `factor` can be used as a fitting parameter. Its name is `factor`. Note that `factor` has a slightly different meaning in *root mode* (see Section 8.1.2 on page 25). Special precautions apply if *factor* is negative. Since the fitting algorithm uses the logarithm of the parameters, it cannot handle negative parameters. If the *factor* is negative, the sign is remembered and the absolute value taken as fitting parameter. The fitting algorithm is not able to cross the boundary between positive and negative paremeters. Thus, is if you anticipate a negative *factor*, you must enter a negative *factor* as start value is the `factor` instruction.

7 Circuit mode: requesting output

After the circuit definition is complete, one or several *computing instructions* may be given in the input file. Somewhere in the input file must be a `freq` instruction that specifies the frequency range for which all computations are to be performed (see section 10.1 on page 39).

The computing instructions fall into three mutually exclusive categories: transfer functions, op-amp stability and noise. Only instructions from one single category may be given in a single run of the program. (This limitation is not a fundamental one, it is mainly there because otherwise the programs control logic would become more complex, and because it usually doesn't make sense to plot the computed results from the three categories together.)

7.1 Transfer functions

7.1.1 Treatment of the input

For a normal voltage input, the voltage at the input node is set to 1 V (at the frequency being computed). For a floating voltage input, the voltage between the input nodes is set to 1 V. For a current input, a current of 1 A is injected into the input node. The *source-impedance* given in the `uinput` or `iinput` statement is ignored for these calculations. If a `factor` instruction is present, the input is multiplied by the given *factor*.

---

the unknown impedance and data measured under the above circumstances can be made consistent by including a `factor 2` instruction which effectively introduces a voltage source of 2 V at the source node (before the 50 Ω source impedance). Another possible application is when the measured data have passed through an inverting stage and thus have 180° phase shift with respect to the simulated model.

7.1.2 Specifying the output format

For any complex function to be computed and plotted, the output format can be specified with an option in the instruction that requests the output. That option, called *coordinates*, can either be empty or a string of the following type:

`:db|abs|re|im|deg|deg+|deg-[:db|abs|re|im|deg|deg+|deg-]` ...

Note that this string must not contain any spaces. It defines the format in which the (complex) transfer function is to be written to the output file and plotted.

`re` and `im` represent the real and imaginary part of the transfer function,
`abs` its absolute value $\texttt{abs} = \sqrt{\texttt{re}^2 + \texttt{im}^2}$,
`db` the absolute value expressed in decibel $\texttt{db} = 20 \log_{10}(\texttt{abs})$ and
`deg` the phase expressed in degrees with $-180° \leq \texttt{deg} \leq 180°$. To avoid ugly phase jumps, the two options
`deg+` and `deg-` also yield the phase in degrees, but with $0 \leq \texttt{deg+} < 360°$ and $-360° < \texttt{deg}- \leq 0$, respectively.

Any combination of these can be specified, and the output file will contain one column for each specified coordinate, in the order given. If *coordinates* is not specified at all, a default of `:db:deg` is assumed.

Example: If you want as output the absolute value of the transfer function for a node called `nsum`, and both absolute value and phase for a node called `nout`, you could enter:

```
uoutput nsum:abs nout:abs:deg
```

7.1.3 Voltage transfer functions

Voltage transfer functions are computed by setting up the input as described in the previous section, and then computing the voltage at the specified nodes. If there is a voltage input, the result is the standard voltage transfer function. If, on the other hand, the circuit has a current input, the computed transfer function has the dimension $1\,\mathrm{V}/1\,\mathrm{A} = 1\,\Omega$ and corresponds to the "transimpedance" of the circuit. Both of these cases will be called "voltage transfer functions" throughout this text. The computation of voltage transfer functions is requested by one or more `uoutput` instructions that have the following general form:

`uoutput all`[*coordinates*]

`uoutput allop`[*coordinates*]

`uoutput` *node*[*coordinates*] [*node*[*coordinates*]] ...

The `all` keyword requests the voltage at all nodes in the circuit to be computed and plotted. The voltage at all op-amp outputs is computed and plotted with the `allop` keyword. The voltage at individual nodes can be requested for computation and plotting with the second form of the instruction. It is possible to combine both forms, if, for example, the absolute value of the voltage at all nodes is desired, plus the phase (in degrees) at the node `no`. The instruction

```
uoutput all:abs no:abs:deg
```

will produce just that output. Note that the two items `all:abs` and `no:abs:deg` must appear in that order to achieve the desired effect.

7.1.4 Current transfer functions

These instructions work similar to the `uoutput` instructions, except that now the output is not the voltage at certain nodes, but instead the current through certain components. For passive components, the current is counted as flowing from *node1* to *node2*, whereas for an op-amp, the output current is computed. For circuits with a current input, the `ioutput` instruction computes a "current transfer function". If the circuit has a voltage input, however, the result is "current flowing for 1 V input voltage" with the unit of $1\,\mathrm{A}/1\,\mathrm{V} = 1/\Omega$, i.e. something like a conductance, although the computed current may actually be supplied by an op-amp and not from the input. The syntax of the required instructions is:

`ioutput all`[*coordinates*]

`ioutput allop`[*coordinates*]

`ioutput` *component*[*coordinates*] [*component*[*coordinates*]] . . .

where *component* now stands for the name of a previously defined component (resistor, capacitor, inductor or op-amp). The option *coordinates* is defined in Section 7.1.2 above. As expected, the `all` keyword requests the current through all components to be computed, the `allop` keyword requests the current from all op-amp outputs to be computed, and individual currents can be chosen by the second form of the instruction. The two forms can be combined in the same way as described at the end of Section 7.1.3.

7.1.5 Maximum permissible input signal

In some complex active filter circuits (e.g. state-variable filters), the voltage at internal nodes can be considerably higher than at the in- and output. If any op-amp output voltage (or current) exceeds the device's limits, the function of the circuit is obviously disturbed. To predict such effects, the `maxinput` instruction computes the maximum permissible input signal, taking into account the maximum output voltage, maximum output current and slew rate of each op-amp in the circuit. These maximum values are defined in the op-amp library for each type of op-amp (see section 12 on page 42), but can be overridden by the user for each individual op-amp in the circuit (see section 6.5 on page 15). The instruction has the following simple form without any options:

`maxinput`

It can be used together with the other "transfer function" output instructions described in this section. The output produced is a single curve representing the maximum input signal in Volts or Ampères, depending on whether a voltage- or current input is specified. The format of the output is the absolute value (corresponding to a `:abs` coordinate specifier). No other output format is supported.

For each frequency, the input limitation due to each op-amp is computed. The minimum of these numbers is then the result to be plotted. During the computation, the op-amp

that causes the limit is identified and printed out (both to the console and the output file). This additional output consists of lines such as

`from` x `Hz onwards input is limited by Umax(` *op-amp name* `).`
`from` x `Hz onwards input is limited by Imax(` *op-amp name* `).`
`from` x `Hz onwards input is limited by SR(` *op-amp name* `).`

for limits being caused by the output voltage, output current or slew rate, respectively. Note that if the permissible input signal is limited by an op-amp's slew rate, the actual filter performance may deviate strongly from the LISO small-signal prediction, because most op-amps need considerable differential input voltages to achieve their stated slew rate.

In fitting, the `mininput` instruction can be specified to put the additional constraint on the fit that the permissible input signal must not fall below a given limit (see section 9.5.1 on page 32).

7.1.6 Input impedance

The input impedance of the circuit can be computed by entering an instruction of the following format:

`zin`[*coordinates*]

where the option *coordinates* is defined in Section 7.1.2 on page 18. It is computed as input voltage divided by current flowing into the input, thus has the unit of $1\,\Omega$, and the result is independent of whether the circuit has a voltage input or a current input. Note that this has nothing to do with the source-impedance defined in the `uinput` or `iinput` instruction. That source-impedance is ignored for the calculation, as for all transfer function calculations.

7.1.7 Op-amp differential input voltages

In a circuit with ideal op-amps and properly connected negative feedback, the differential input voltage for an op-amp (i.e. the voltage between the non-inverting and inverting input) will always be zero. However, for real op-amps with finite gain, this voltage will be finite. In order to check whether this effect will disturb a given circuit, these differential voltages can be computed and plotted by instructions of the following format:

`opdiff all`[*coordinates*]

`opdiff` *op-name*[*coordinates*] [*op-name*[*coordinates*]] ...

where the option *coordinates* is defined in Section 7.1.2 on page 18. The *op-name* must be the name of a previously defined op-amp. The units are Volts, for a given input of either 1 Volt or 1 Ampère. As expected, the `all` keyword requests the differential input voltage for all op-amps to be plotted, whereas individual op-amps can be selected with the second form. The remark in Section 7.1.3 on page 18 about combining both forms is applicable here as well.

7.2 Op-amp stability

In many circuits it is difficult to determine beforehand whether an op-amp will be stable or will oscillate. In order to understand and predict this behaviour, the "stability function" for some or all op-amps in a given circuit can be computed. The required instructions are

`opstab all`[*coordinates*]

`opstab` *op-name*[*coordinates*] [*op-name*[*coordinates*]]

where the option *coordinates* is defined in Section 7.1.2 on page 18. The *op-name* must be the name of a previously defined op-amp. As expected, the `all` keyword requests the stability function for all op-amps to be plotted, whereas individual op-amps can be selected with the second form. The remark in Section 7.1.3 on page 18 about combining both forms is applicable here as well.

The stability function needs to be computed separately for each op-amp in question. That means that the total computation time is multiplied by the number of op-amps for which the stability function is computed (as opposed to all other possible outputs of LISO, which can be computed simultaneously). For each op-amp, the computation is done as follows: No voltage or current is connected to the circuit input, i.e. it makes no difference whether a voltage input or current input was specified. The specified source impedance is assumed to be connected between the input node and GND. The op-amp in question is "disconnected" from the circuit, i.e. it no longer has its normal functions. All other components (including other op-amps) retain their normal functions. A fixed voltage of 1 V is applied to the node where the op-amp's output was connected. The result of the computation is the transfer function from the op-amp's output to its differential input voltage multiplied with the op-amp's open loop gain. Thus, the result is the open-loop gain of the feedback loop consisting of the op-amp in question and the rest of the circuit (see also Section 13.3 on page 50). The usual stability conditions for control loops can be applied to this open-loop gain, e.g. the phase delay must be less than 360° at the unity-gain frequency[3]. Note that obviously the result can only be as accurate as the op-amp's open-loop gain is modelled (which is taken from the op-amp library, with possible "overrides" by the user).

7.3 Noise calculations

The program knows about three different types of noise sources in the circuit: Johnson noise of resistors ($\widetilde{U}_n = \sqrt{4kTR}$), and voltage and current noise of op-amps (taken from the op-amp library, with possible "overrides" by the user). The op-amp's voltage noise is assumed to be applied between its input nodes. The current noise is assumed to be identical for the inverting and non-inverting input[4] (for some more detail see Section 13.2 on page 48). However, because both op-amp inputs will usually be connected to different impedances, the current noise densities of both inputs are treated

[3] This corresponds to the usual 180°, where *negative* feedback is implicitly assumed.

[4] This is true for most voltage-feedback op-amps, but not for current-feedback op-amps, which at present cannnot be handled properly by the program, anyway.

as separate noise sources, such that one op-amp in total has three noise sources: one voltage noise and two current noise sources.

The resistor noise itself is frequency-independent, whereas the op-amp noise may have $1/f$-corners (i.e. the noise increases below a certain frequency). The effective noise at any node of the circuit (such as the output) will in general always be frequency-dependent, because the circuit has frequency-dependent impedances, gains, etc.

For the computation of noise, the distinction between `uinput` and `iinput` is ignored, since no input signal is assumed. The *source-impedance* given in the `uinput` or `iinput` instruction is assumed to be connected from the input node to ground. It will affect the gain of noise contributions from their source to the output. The impedance itself is considered noise-free, i.e. no Johnson noise is computed for it. If you want to compute the source impedance's Johnson noise, you must explicitly enter it as a resistor.

There are two types of instructions that describe which noise contributions should be computed and plotted. A `noise` instruction requests a noise contribution to be plotted. It has the format

`noise` *node* `sum`|`all`|`allr`|`allop`|*noise-source* [`sum`|`all`|`allr`|`allop`|*noise-source*] ...

The noise is computed at the node called *node*. Note that in one run of LISO, noise contributions at only one node can be computed. Specifying `all` causes all noise contributions in the circuit to be plotted separately. The keywords `allop` cause all op-amp noise contributions and `allr` all resistor noise contributions to be plotted, respectively. Specifying `sum` causes an additional output, the sum of all specified noise sources (see below). An item *noise-source* can either be the name of a resistor, the name of an op-amp, or the name of an op-amp with the following suffix:

`:u|+|-` [`u|+|-`]

If only the name of the op-amp is given without any suffix, all three noise contributions are plotted. Otherwise, to select only one or two of its noise sources, the suffixes indicate which of the three noise sources are to be plotted. The letter `u` stands for the voltage noise, `+` represents the current noise of the non-inverting input and `-` represents the current noise of the inverting input. For example, if you want to plot the noise contributions of a resistor `r1` and both current noise contributions of an op-amp called `op2` at a node called `nout`, you should enter

```
noise nout r1 op2:+-
```

The suffixes `:u|+|-` [`u|+|-`] can also be entered after the keywords `all` or `allop`.

Plotting all individual noise contributions of a complex circuit will produce a messy plot with many lines. The option `sum` of the `noise` instruction allows to plot the sum of several or all noise contributions. In order to specify which noise contributions are to be added, there is an additional instruction called `noisy`. Its format is:

`noisy` `all`|`allr`|`allop`|*noise-source* [`sum`|`all`|`allr`|`allop`|*noise-source*] ...

The keywords `all`, `allop` and any individual op-amp name can again be followed by the suffixes (`:u|+|-` [`u|+|-`] dots) with the same meaning as above.

Note that the `noisy` instruction has an effect only if the `sum` keyword is included in the `noise` instruction. It can be used to switch on/off individual noise sources as

far as their contribution to the sum is concerned. Note also that all noise sources that are included in the `noise` instruction, i.e. those that are plotted individually, are automatically considered "`noisy`", i.e. they are always included in the sum.

In case of doubt, please examine the end of the output file, where all noise sources are listed to find out what was included in the sum. As a simple example, the following two instructions

`noisy all`
`noise` *output-node* `allop:u sum`

cause a plot of the individual contributions of all op-amp voltage noise sources and the sum of all noise sources (i.e. including op-amp current noise and resistor noise).

Note that the computed noise always has the unit of $V/\sqrt{Hz}$, i.e. it is a linear spectral density. It is computed by multiplying the spectral density at the noise source with the transfer function from that source to the specified node. It is *not* referred to the circuit input. If you want that, you must divide the computed noise at the output by the transfer function from input to output (which can of course be found in a separate run of LISO).

8 Root mode

The second main operating mode of LISO is called root mode. It is much simpler than the circuit mode. Its purpose is to compute or fit a frequency response defined by poles and zeroes of the transfer function. Some applications include:

- To compute a frequency response from poles and zeroes (e.g. coming from a filter table) as input to the LISO circuit mode fitting function.
- To fit measured data (e.g. from a pendulum-type control element) in order to determine pole frequencies, Q's, etc.
- To fit some computed output from LISO circuit mode in order to describe the resulting frequency response in terms of poles and zeroes.

In root mode, the input file consists of three kinds of instructions:

Transfer function definition The transfer function is defined by one or more `pole` or `zero` instructions. Additionally there may be `factor`, `ffactor` or `delay` instructions, one of each kind at most. These may come in any order, but must be completed before fitting instructions appear.

Computing instruction There must be exactly one `freq` instruction, which describes the frequency range of the plot (unless a fit is done and the frequency range is taken from the data file). There is only one computing instruction, `tfoutput`. It is used to define the type of output desired. It may be omitted, if the default is acceptable.

Fitting instructions These instructions are optional and used only if an optimization is desired. If they are absent, the program will only compute the given transfer function. For an optimization, there must be one or more `param` instructions, optionally followed by `sparam`, `pparam` or `rparam` instructions. These define the parameters to be varied in the optimization. There must also be exactly one `fit` instruction which describes the input file for the fitting procedure, e.g. the measured or desired frequency response to be approximated and its file format (see Section 9.3 on page 27).

8.1 Transfer function definition

The transfer function has the following general form:

$$H(\hat{f}) = A_0 \frac{z_0(\hat{f}) \cdot \ldots \cdot z_{\mathrm{nz}}(\hat{f})}{p_0(\hat{f}) \cdot \ldots \cdot p_{\mathrm{np}}(\hat{f})} \exp(-2\pi \mathrm{i}\, \hat{f}\, t_{\mathrm{delay}}), \tag{1}$$

where A_0 is the overall gain (called `factor` in LISO), t_{delay} is the delay time (called `delay` in LISO), $\hat{f}$ is the frequency divided by `ffactor` (that corresponds to multiplying all pole- and zero-frequencies by `ffactor`). Note that all frequencies in LISO are physical frequencies f, and *not* angular frequencies $\omega = 2\pi f$. The poles $p_k(f)$ and zeroes $z_k(f)$ can either be singles or conjugate-complex pairs:

$$p_k(f) = 1 + \frac{\mathrm{i}f}{f_k} \quad \text{single} \tag{2}$$

$$p_k(f) = 1 + \frac{\mathrm{i}f}{f_k Q_k} - \frac{f^2}{f_k^2} \quad \text{c.c. pair} \tag{3}$$

with coprresponding expressions for z_k.

8.1.1 Poles and zeroes

Poles and zeroes of the transfer function are entered by the `pole` and `zero` instructions:

`pole`|`zero` *frequency* [Q]

If Q is present, a conjugate-complex pair is assumed, otherwise a single pole/zero. The order in which poles and zeroes are entered is unimportant if no fit is performed. For a fit, the program enumerates the poles and zeroes separately in the order of entry, starting with 0. Thus the first pole is called `pole0`, the second one `pole1` etc. The zeroes are separately numbered `zero0`, `zero1` etc. Note that one pole (or one zero) for the program can either be single or a c.c. pair, depending on whether a Q was entered in the definition. Both *frequency* and Q must be positive. At present there is no possibility to directly enter a zero of the form

$$z_i(f) = 1 - \frac{f^2}{f_i^2} \quad \text{c.c. pair.} \tag{4}$$

This can, however, be approximated by entering a huge value for Q, e.g. 10^6.

8.1.2 Overall factor

An overall factor (A_0 in equation (1)) can be entered by the instruction

`factor` *factor*

It can be either positive or negative. Since for a fit all parameters must be positive, a negative `factor` is internally multiplied by -1, and the sign is remembered. Since all other parameters must always be positive, the absolute value of `factor` is used internally for all computations, e.g. if `factor` is a fitted parameter and other parameters depend on it.

8.1.3 Frequency scaling

The frequency of all poles and zeroes (and the delay time, if present) can be scaled by a `ffactor` instruction:

`ffactor` *scale-factor*

This allows, for example, to enter poles and zeroes directly from filter tables, which correspond to a nominal corner frequency of 1 Hz. By entering the real corner frequency as *scale-factor* one obtains a filter transfer function with the desired corner frequency. The *scale-factor* must be positive.

8.1.4 Delay time

Some transfer functions include a delay time t_{delay}. Its effect is a phase shift that grows proportional to the frequency. It can be entered with the `delay` instruction

`delay` *delay-time*

If there is a `ffactor` instruction, the effective delay time is appropriately scaled internally.

8.2 Requesting output of the transfer function

As described in Section 10.1 on page 39, the input file must contain exactly one `freq` instruction. In addition there may be a `tfoutput` instruction which can be used to specify the coordinates in which the transfer function should be plotted. Its format is

`tfoutput` *coordinates*

The string *coordinates* has exactly the same format and meaning as defined in Section 7.1.2 on page 18. As described there, the default is `db:deg`, which is also assumed if no `tfoutput` instruction is there at all.

9 Fitting

9.1 Introduction

The fitting function of the LISO program is one of its most important features. It works as follows: In addition to the complete circuit desciption, a data file must be given that contains the same type of data that is computed by the simulation, e.g. a frequency response. For simplicity, we will call this data file the "measured data", although in many applications it may as well come from some other simulation (e.g. an ideal n-pole Butterworth filter response or some other ideal desired function).

In general, the measured data will not coincide with the simulated data, e.g. due to unknown component values or non-ideal op-amp characteristics. The purpose of the fitting function is to vary some parameters (such as component values) of the simulated circuit until the best possible coincidence between measured data and simulated data is obtained.

Some applications include:

- In the design of active filters, the standard design techniques will usually produce a circuit that only approximately has the desired ("ideal") frequency response, because op-amps at higher frequencies are no longer ideal. With LISO's fitting function it is possible to find modified component values that yield a circuit with a frequency response as close as possible to the ideal response, *taking into account the non-ideal op-amp characteristics.*
- Even at low frequencies where op-amp deficiencies are not yet important, it is often very difficult to find suitable component values for a given circuit topology and frequency response. "Textbook" solutions often imply arbitrary boundary conditions (such as equal resistors) just because of the mathematical complexity of a general solution. LISO's fitting function allows the user to find component values without these boundary conditions and helps the user to optimize them for a given purpose (such as lowest noise or maximum bandwidth).
- In some situations certain parameters of a circuit are not well known, such as photodiode capacitances or other parasitic capacitances or inductances. If a frequency response has been measured, and most components of the circuit in question are known, it is possible to find the unknown parameters by the fitting function. Once these are known, the circuit can be optimized for a certain purpose.
- If the characteristics of an op-amp are not well known, but some data exist about its frequency response (e.g. from a data sheet or from an actual measurement in a simple circuit), it is possible to find the op-amp parameters that most closely match the frequency response. These parameters can then be entered into the op-amp library and used for subsequent simulations or optimizations of more complex circuits.

9.2 Function to be fitted

At present, only "transfer functions" (see Sections 7.1.3 . . . 7.1.7) can be used for fitting. Usually an input file used for fitting will contain only one of the instructions

`uoutput`, `ioutput`, `zin` or `opdiff`. The transfer function requested by that instruction is then used for the fit. If there are several such instructions, the first output instruction found is used as function to be fitted. The output format of the transfer function to be fitted needs not to be identical to the format of the data file, e.g. it is possible to have a data file in the form of dB and degrees and request the output as real and imaginary part. Both data formats will be converted into an internal representation (real and imaginary part if the phase is known, and absolute value otherwise) for the fit, and the plot will use the requested output format of the transfer function.

9.3 Data file

If a fit is to be performed, there must be exactly one instruction in the input file with the following format:

`fit` *file-name* `reim`|`dbdeg`|`absdeg`|`db`|`abs` `abs`|`rel`|`semi`|`sdev` [`plotfirst`]

Apart from the `fit` keyword, there must be three items on this instruction. The first one is the *file-name* of the data file to be fitted. The second one describes the type of data in that file, and the third one the weighting to be employed. The option `plotfirst` causes all functions to be plotted twice: First with the startvalues for all parameters (i.e. before the fit), and then in the end with the best-fit values for all parameters. If the option `plotfirst` is not present, the functions will be plotted only once, using the best-fit parameters.

Currently five different types of data file are supported (see below). All of them must be ASCII files. One line in the data file describes one data point. Empty lines and lines beginning with '`#`' are ignored. The first number in each line (i.e. the first column in the data file) must be the frequency. After the frequency either one or two numbers (the data) are expected. Finally there may be an optional extra number describing the weight. This extra number needs not to be present in all input lines and is described in Section 9.3.1.

The data to be fitted can be of two basic types: The first are *absolute values*, which must be non-negative. They can be entered either directly, or as decibel (dB). Secondly, if phase information is available, the data can be complex numbers. They can be entered either directly (as real and imaginary parts), or as combinations of absolute value and phase (in degrees).

Accordingly, the five supported types of data files are:

- `reim` The real and imaginary part of the complex function.
- `dbdeg` The absolute value of the complex function (in dB) and its phase (in degree).
- `absdeg` The absolute value of the complex function and its phase (in degree).
- `db` The absolute value of the complex function (in dB).
- `abs` The absolute value of the complex function.

Data points with absolute value 0 are ignored, and a warning message is printed.

9.3.1 Weighting

The last item of the `fit` instruction describes the *weighting* of the data points. The fitting algorithm (described in more detail in Section 14 on page 51) minimizes a real and non-negative function conventionally called χ^2 which is defined as follows:

$$\chi^2(p_1, \dots, p_m) = \sum_{i=1}^{n_{\text{data}}} w_i \, |y_i - \tilde{y}(f_i, p_1, \dots, p_m)|^2 \tag{5}$$

where n_{data} is the number of the measured data points y_i, $\tilde{y}$ is the computed (simulated) frequency response, f_i is the frequency of the ith data point, $p_1, \dots, p_m$ are the parameters to be fitted, and w_i is the weight of each data point to be discussed here. First note that all data points are internally converted to complex numbers or absolute values before any calculations are done. No matter in which format the data is read from the data file or plotted (such as dB, absolute value, etc.), both y_i and $\tilde{y}$ are always taken as either absolute values or complex numbers. In other words, the fitting algorithm never uses dB or degrees to compute χ^2. Now, there are several common possibilites to define the weights w_i.

- All weights are the same, $w_i \equiv 1$. This is often called *absolute weighting*. The χ^2 function is dominated by those data points with the largest absolute values, which is usually not the most useful approach for the typical applications of LISO. As an example, the difference between 10 and 10.1 counts as much as the difference between 0.1 and 0.2.

- In the so-called *relative weighting*, one sets $w_i = 1/|y_i|^2$. The effect is that the relative error in all data points contributes equally to the χ^2 function, independent of the absolute value of the data point. If no individual weights are known for the data points, this is usually a better approach than absolute weighting. With this approach, the difference between 10 and 11 counts as much as the difference between 0.01 and 0.011, or, in other words, a quotient expressed in dB counts always the same, independent of the absolute value of the data point.

- A compromise between the last two alternatives is to set $w_i = 1/|y_i|$. I call this *semi-relative* weighting. In many typical applications of LISO this is the most sensible weighting.

- In the special case that the data come from an actual measurement with measurement errors that are known and have a normal distribution, the statistically correct weights are given by $w_i = 1/\sigma_i^2$, where σ_i is the standard deviation of the data point i. If such data is available, use it. Only then the estimated errors of the fitted parameters can have absolute meanings.

Additionally the user may wish to assign extra weights to certain data points, either because they are considered very important, or else because they are considered unreliable or unimportant. In LISO, this effect can be obtained by entering an extra column in the data file which is interpreted as *weight multiplier*. This extra number needs not

to be present for all data points. If it is present, the computed weight is multiplied by that number. The effect is the same as if the data point in question were present several times in the data set. For example, a weight multiplier of 10 is equivalent to having 10 identical data points. A weight multiplier of 0 can be used to effectively exclude a data point from the fitting procedure[5].

Now the meaning of the keywords `abs`|`rel`|`semi`|`sdev` in the `fit` input line should be easily understandable:

`abs` All weights w_i are set to unity. If a data point has a weight-multiplier, the weight is set to the weight-multiplier instead.

`rel` The weights are computed from the data points as $w_i = 1/|y_i|^2$. If $|y_i| = 0$, then the weight is set to unity. If a data point has a weight-multiplier, the weight is multiplied with it.

`semi` The weights are computed from the data points as $w_i = 1/|y_i|$. If $|y_i| = 0$, then the weight is set to unity. If a data point has a weight-multiplier, the weight is multiplied with it.

`sdev` The extra column after the data (which is interpreted as weight-multiplier in the above three cases) is now interpreted as standard deviation σ_i, and the weight is set to $w_i = 1/\sigma_i^2$. If $\sigma_i \leq 0$ or σ_i is missing, w_i is set to unity and a warning message is printed.

9.4 Parameters

A *parameter* in this context is a non–negative variable that influences the output of the simulation. Typical examples are component values, pole frequencies, etc. During the fit, the parameters are varied by the fitting algorithm, in order to bring simulated data and measured data together as closely as possible. The basic instruction to define a parameter is the `param` instruction with the following syntax:

`param` *parameter-name* *lower-limit* *upper-limit*

Note that there must be a separate `param` instruction for every parameter. The limits *lower-limit* and *upper-limit* must both be positive numbers. They restrict the range in which the fitting algorithm is allowed to vary the parameter. Note that all parameters defined by `param` instructions must have been previously defined in the circuit (or transfer function) definition. The value from the definition is taken as starting value for the fitting algorithm.

In circuit mode, the *parameter-name* can be any of the following:

- The name of a passive component (resistor, capacitor, inductance or mutual inductance). The parameter itself is then the value of that component (or the coupling factor in the case of a mutual inductance).

[5] This is obviously computationally inefficient and should not be used if computation time is a problem.

- The name of an op-amp, followed by a colon (:) and one of the op-amp characteristics described in Section 12 on page 42 (e.g. `op1:gbw` or `op3:pole0`). The parameter is then that specified characteristic of the op-amp.

In root mode, the *parameter-name* can be any of the following:

- The word `pole`, followed by the number of the pole (according to the entry of poles, see Section 8.1.1 on page 24), followed by either `:f` or `:q`. This chooses a pole frequency or pole Q as parameter. As an example, the frequency of the first pole is called `pole0:f`. (Note that all counting starts from zero in LISO.)
- The same as above with the word `zero` instead of `pole`.
- The word `factor`, choosing the overall factor A_0 as parameter.
- The word `ffactor`, choosing the overall frequency factor A_0 as parameter. Note that it is a bad idea to choose *all* pole frequencies *and all* zero frequencies as parameters, if `ffactor` is also chosen as parameter. The system would then be underdetermined.
- The word `delay`, choosing the delay time t_{delay} as parameter.

9.4.1 Dependent parameters

In some situations, several parameters need to be varied together, but not independently. One example is a circuit with several identical op-amps. If a certain op-amp characteristic, such as its gain-bandwidth-product, is used as fitting parameter, one may want to vary that parameter for all (or some) op-amps simultaneously. Another example is a filter circuit whose DC gain is determined by the ratio of two resistors. If that DC gain is known and fixed, it is desirable to vary only one of the resistors as true parameter and let the other follow with the fixed ratio[6]. The instructions to define these dependent parameters are

`sparam` *parameter-name*

`pparam` *parameter-name* [*factor*]

`rparam` *parameter-name* [*factor*]

parameter-name has the same meaning as in Section 9.4 above. A `sparam` ("simultaneous" or "same") instruction defines a parameter that is equal to the corresponding independent parameter. A `pparam` ("proportional") instruction defines a parameter that is linearly proportional to the corresponding independent parameter, with the factor given. Finally, a `rparam` ("reverse proportional") instruction defines a parameter that is inversely proportional to the corresponding independent parameter, with the

[6] If such a priori information is known, it is always advantageous to give it to the fitting algorithm. If, in the example given, two independent parameters were used, the fitting algorithm would maintain the ratio of both resistors very close to the known ratio, if the measured data has the given DC gain. This is, however, much less efficient and reliable than *telling* the fitting algorithm the known ratio (and thus reducing the number of independent parameters by one).

'factor' given being the constant product of both parameters. These three instructions can be arbitrarily combined or mixed. However, they must immediately follow a `param` instruction that defines the corresponding independent parameter.

If the factor is omitted in an `pparam` or `rparam` instruction, it is computed from the starting values of both parameters. If the factor is given, but in disagreement with the two starting values, the starting value of the dependent parameter is adjusted according to the given factor, and a warning message is printed.

As an example, consider the following instructions:

```
param r1 100 10k      # first independent parameter
sparam r2             # depends on r1, r2=r1
pparam r3 5           # depends on r1, r3=5*r1
rparam c1 0.001       # depends on r1, c1=0.001/r1
param op1:gbw 3M 10M  # second independent parameter
sparam op2:gbw        # same as op1:gbw
```

This defines two independent parameters (`r1` and `op1:gbw`) which may be varied between 100 Ω and 10 kΩ and between 3 MHz and 10 MHz, respectively. The dependent parameter `r2` is always equal to `r1`. The dependent parameter `r3` is always equal to 5·`r1`, whereas the dependent parameter `c1` is always equal to 0.001/`r1`. The gain-bandwidth products of `op1` and `op2` are varied together (typical for a dual op-amp).

9.5 Additional fitting constraints for optimizing the dynamic range

In the design of filter circuits, there is often a multitude of possible 'solutions' (i.e. sets of component values that provide the desired transfer function). With the LISO instructions introduced so far, such sets can be found and the resulting circuit can be analyzed for op-amp stability, noise, maximum input signal etc.

Sometimes it is desirable to optimize the dynamic range of the circuit. As dynamic range we understand here the ratio of maximum permissible input signal to the output noise generated in the circuit. (Since we assume the transfer function to be fixed, minimizing the output noise is equivalent to minimizing the input-referred noise.)

Usually the dynamic range is limited at the upper end by the maximum output voltage and/or output current that the op-amp's can produce. A typical specification might be that the output may swing ±12 V in the frequency range of interest, i.e. the circuit is not limited by either overcurrent from op-amps output or higher voltages at internal nodes due to suboptimal gain distribution. This condition sets lower limits to resistor values.

At the lower end, the limit is noise. The noise can be reduced by reducing resistor values (thereby reducing both Johnson noise and the effect of op-amp current noise), but also by redistributing the gain between the stages of a multi-stage filter.

Optimizing the dynamic range hence involves reducing resistor values as much as possible without overloading any op-amp, while always maintaining the desired transfer

function. Since in many circuits transfer function, noise and op-amp load are connected in a very complicated way, this optimization rarely finds the true optimum, if done 'manually'.

Hence two new instructions have been added to LISO which allow to (at least partially) automatize the procedure. They work by putting additional constraints on a fit (in circuit mode). The `mininput` instruction requests that the allowable input signal must be at least a given value. The `minnoise` instruction aims for a given *rms* noise value at the output. They are explained below. Some hints for their application are given in Section 9.8 on page 36. Note that their implementation is rather new and still experimental. Any feedback on the success or failure of these instructions is very much desired by the author.

9.5.1 Input range constraint

In various circuit, the permissible input signal may become unacceptably small due to op-amp overloads at internal nodes. If there are enough free parameters, this situation can often be avoided by proper choice of parameters. The `mininput` instruction helps finding such parameters. Its format is:

`mininput` *value min-freq max-freq*

It puts the following additional constraint on the fit: The permissible input signal (as it would be computed by the `maxinput` instruction, see Section 7.1.5 on page 19) must be bigger than *value* for all frequencies between *min-freq* and *max-freq*. In a typical application, *min-freq* and *max-freq* specify the filter's passband, and *value* is the desired output range (e.g. 10 V) divided by the filter's passband gain.

The instruction is implemented in such a way that only those frequencies between *min-freq* and *max-freq* are checked where a data point is given. All such data points contribute equally to a 'punishment' function, which makes the fit look worse, if the constraint is violated, thereby forcing the fitting algorithm back into the allowed region of parameter space. A consequence is that the degree of 'punishment' applied depends on the number of data points between *min-freq* and *max-freq* and can be made stronger by including more data points for those frequencies.

The `mininput` instruction works best with starting values that already describe a 'solution' (i.e. a set of parameters which produce the desired transfer function). Otherwise the `mininput` constraint may prevent LISO from finding a solution in the first place (see also Section 9.8 on page 36).

Because the χ^2 function is modified in a highly nonlinear way by the `mininput` instruction, the correlation matrix and parameter error estimates printed at the end of the fit are usually wrong and useless.

It is often interesting after a fit to look at the optimized transfer function and the allowable input signal at the same time. This can be achieved with instructions like:

`uoutput` *node*`:abs:deg`
`maxinput`

9.5.2 Noise minimization

The `minnoise` instruction is used to add another 'goodness of fit' criterion to the fitting algorithm: Minimize the noise at a certain node. The syntax is:

`minnoise` *node min-freq max-freq aim*

The noise is computed at the node named *node*, e.g. the output. It is computed as rms value between the frequencies *min-freq* and *max-freq*. Depending on the application, *min-freq* and *max-freq* may specify either the passband or the stopband of the filter, or any other frequency range where the noise is to be minimized.

The noise is computed only for those discrete frequencies between *min-freq* and *max-freq*, where a data point is in the data file. Increasing the number of data points in that frequency range will increase the accuracy of the noise computation, but not the scale factor.

The value *aim* is very important. It specifies the desired rms noise in the given frequency band (with the unit Volt). If the computed noise is below *aim*, no penalty is added to the χ^2 function. If, on the other hand, the computed noise is bigger than *aim*, χ^2 is increased in such a way, that the χ^2 for an otherwise perfectly fitting circuit is at least

$$\chi^2_{\text{minimum}} = 2 \log \frac{\text{Noise}_{\text{computed}}}{\text{Noise}_{\text{aim}}}. \tag{6}$$

If the aim is unreachable, the fitting algorithm will try to find a compromise between lowest noise, non-fitting transfer function and violation of other constraints (such as `mininput`). A possible choice for *aim* is around 0.5 times the reachable minimum. The compromise found will depend on the number of data points used for the fit and in particular on the weighting option used. Some experimentation will in general be necessary to find optimum results. The proposed *aim* of 0.5 times the reachable minimum corresponds to a minimal χ^2 of around 1.4, which usually means only a small deviation of the desired transfer function.

The noise is computed for all components declared 'noisy' by a `noisy` instruction (see Section 4 on page 22). Most often this will be a simple `noisy all` instruction.

The `minnoise` instruction should only be used together with other constraints, which place a lower limit on resistor values (either direct parameter limits or, more interesting, the `mininput` instruction). Otherwise the algorithm will propably reduce the resistors to unrealistic small values.

Because the χ^2 function is modified in a highly nonlinear way by the `minnoise` instruction, the correlation matrix and parameter error estimates printed at the end of the fit are usually wrong and useless.

9.6 Rewriting the input file with best-fit parameters

Once a fit has been performed, one will often want to do additional computations in separate runs of LISO, using the parameter values found by the fit. For this purpose,

the `rewrite` instruction causes a copy of the input file to be written, where all fitted parameters are replaced by the best-fit values. The required syntax is:

`rewrite` [*file-name* | `same`]

If *file-name* is given, the result is written to that file. Otherwise, the filename is constructed by appending '`_f.fil`' to the basename. If, for example, the input file was `test.fil`, the rewritten file will be called `test_f.fil`. Note that the name of the rewritten file *must not* be identical to the input file (because the input file is copied line by line, except for those lines that contain fitted parameters). If the *file-name* given is identical to the name of the input file, both files will be lost.

If the fitted parameters are to be written to the input file, use the `rewrite same` instruction. It first writes to temporary file, then saves the original input file (with the name '`_saved.fil`') and finally renames the temporary file to the original input file name.

9.7 Output of the fitting procedure

Here is a typical output of LISO's fitting function:

```
nfunc  Stat       ssq        r1        r3        r4
    1 Start     1.978       170    1.075k        40
    5  Splx     1.528     247.9    1.075k     35.01
   10  Splx     1.364     212.6     975.8     22.89
   15  Splx    0.9682     265.3    1.338k     25.88
   17  Splx    0.5246     204.7    1.118k     32.84
   21  Splx    0.3366     223.1    1.074k     26.06
   23  Splx    0.1327     184.3     902.5     34.14
   33  Splx   0.09768     187.1     884.4     36.17
   37  Splx   0.08589     187.6     927.7     35.78
   43  Splx   0.07336     179.8     879.7     37.17
   60  Sout   0.06952     175.9     856.1     38.95
   72  Marq   0.06947     176.4     858.8     38.74
   72  Mout   0.06947     176.4     858.8     38.74
   76  Splx    0.3901     166.6_    777.9     37.21
   78  Splx    0.1328       173     887.6     38.22
   80  Splx     0.103     177.5     849.4     40.52^
   86  Splx   0.08596     179.4     849.7      38.4
   90  Splx   0.07574     175.1     866.8     38.68
  121  Sout   0.06957     175.4     853.5     39.07

Correlation matrix
           r1       r3       r4
     r1      1
     r3   0.946        1
     r4  -0.946   -0.935        1
```

```
Best parameter estimates:
r1 =  176.37138 +- 5.083 (2.88%)
r3 =  858.84215 +- 26.1 (3.04%)
r4 =  38.740506 +- 1.608 (4.15%)
```

There are several interesting points to note:

- Because the fitting algorithm involves some random initialization functions (see Section 14.1.4 on page 53), two separate runs of LISO with identical input files will usually produce slightly different output. The end results should, however, be the same. If ever significantly different end results should be obtained with identical input files, please contact the author.

- The first part shows the progress if the fit. The individual columns are:

  nfunc How often the χ^2 function (see equation 5 on page 28) was computed.

  Stat The status of the fitting algorithm. `Start` is the entry point, `Splx` a Simplex step, `Sout` a final Simplex step, `Marq` a Marquardt step and `Mout` a final Marquardt step (see Section 14 on page 51 for an explanation).

  ssq The present value of χ^2 ("sum of squares"). Note that it is *no* bug if the first few Simplex steps after a Marquardt step have a higher χ^2 than the previous minimum (see Section 14.1.4 on page 53).

  r1 ... These are the present values of all independent parameters. An appended underscore _ indicates that the parameter is running against its lower limit, whereas an appended caret ^ indicates running against the upper limit.

- Next comes the correlation matrix. It is an indication of how strongly any two parameters are correlated. The correlation of each parameter with itself is trivially always unity. The correlation between two different parameters can be between -1 and 1. The correlation matrix is the most important diagnostic tool if a fit produces unexpected results. A correlation near 1 or -1 between different parameters almost always indicates some problem (such as too many parameters). See also Sections 9.8 on the following page and 14.5 on page 58. To simplify spotting such problems, any correlations with absolute value larger than 0.95 are printed with a special message after the matrix.

- Finally there are the best parameter estimates. Special care must be taken in interpreting the printed tolerances. *If and only if* the data were measured data that included standard deviations (`sdev` file format) *and* those measurement errors are normally distributed, then these are also standard deviations for the best-fit parameters, but even then the correlation matrix must be taken into account for proper interpretation (see any statistics textbook such as [8]). In the usual case of data *without* real standard deviations, the printed tolerances should only be seen as indications how critical each parameter is in relation to the other parameters. Extremely huge tolerances (sometimes even "Infinity" printed as `InfG`) usually come together with highly correlated parameters and indicate problems such as too many parameters etc.

9.8 Hints for fitting

Obtaining a satisfactory fit is a complex process that cannot fully be automized. It needs some experimentation by the user and is often not successful in the first attempt. Here are some hints for a few common situations:

Bad fit This shows up most clearly by a drastic discrepancy between "fitted" curve and data curve in the final plot. Another indication is a huge[7] value of χ^2. Possible causes include:

- Something is wrong in the model. One possibility to check the model is to examine the output file, at the end of which a copy of the input is written as it was interpreted by LISO. Also helpful is the `plotfirst` option of the `fit` instruction (see Section 9.3 on page 27) which plots a computed curve for the model using the starting values of all parameters. One common error is a wrong phase (off by 180°).
- Model and data file don't fit together. Is the correct output signal computed? Is the data file format entered correctly (e.g. dB and degrees vs. real and imaginary part)? Here also the `plotfirst` option may help.
- Did any parameter run into a limit?
- Maybe there are too few parameters (not enough degrees of freedom)? You may try to add additional parameters, such as more variable components, poles or a variable overall factor.
- In a complex problem (e.g. more than 4 parameters) the fit may fail even with correct model and data, if the starting values are too far away from the solution. This is a common and tricky situation with no general remedy.

  Of course the starting values should be estimated by the user as good as possible. It may help to try a few combinations manually (i.e. by a simulation without the `fit` instruction) to get a feeling for the appropriate orders of magnitude.

  Sometimes it helps to let the fit algorithm first run with only a partial set of parameters, then copy these best estimates into the input file (or use the `rewrite` instruction), modify the selection of parameters and iterate the procedure. With creative experimentation it is almost always possible to find a solution, if it exists at all. The "art" here lies in the selection of parameters to fix or vary.

  One common problem is to find appropriate component values for active filters, when an ideal frequency response is given as data. The author found the following procedure practical: Set all resistors to a convenient value, such as 3 kΩ. Let the `fit` vary all capacitors. If necessary, vary some (as few as possible) resistors as well. Once the fit is good, choose the closest standard values for the capacitors and let now vary only the resistors. Once a solution is found, check the maximum input signal, the noise and other

[7] With the most common weighting options and data sets, a value of $\chi^2 \lessapprox 1$ indicates a reasonable fit, whereas values of 100 or bigger hint towards a failed fit.

figures of merit and modify starting values accordingly. After a few iterations often good results are obtained, which are very hard to find otherwise.

- The *weighting* of the data points may have an enourmous influence on the success of the fit. If for example the measured data is very noisy for low amplitudes, a *relative* weighting may cause a funny behaviour of χ^2 for very wrong parameters, with spurious local minima and other weird effects. Try different weighting options.
- In rare extreme cases it may be necessary to change the behaviour of the fitting algorithm, which can be controlled by the constants named "`SIM_...`" and "`MAR_...`" in the initialization file. This should, however, only be considered when all other possibilities have been thoroughly exhausted and preferably in contact with the author.

Correlated parameters This is another common problem with symptoms such as entries near 1 or −1 in the correlation matrix, and/or unreasonably large "standard deviations" for the best parameter estimates. It is generally caused by having too many degrees of freedom in the model, such that multiple equivalent solutions exist[8]. Luckily here the remedy is simple: reduce the number of independent parameters. Depending on the situation, this may be achieved by either simply fixing one or more parameters (such as disabling the `param` instruction with a `#` comment indicator) or using dependent parameters (see Section 9.4.1 on page 30).

Advanced fitting with `mininput` and `minnoise`

Those two instructions can be very powerful, if used correctly. They allow the user to optimize a given circuit for maximal dynamic range, i.e. ratio between maximal allowable signal and noise. They are rather new and their implementation is still experimental. There is not yet much experience with their application. Here are some hints from the first experiments (designing active filters):

- As usual, a data file with the desired frequency response is needed and can be produce by LISO in the 'root mode'. The number of data points in the frequency range of interest plays some role in the effect of the `mininput` instruction and in the accuracy of the noise calculation for the `minnoise` instruction, but is usually not critical.
- It is best to first find a 'solution' (i.e. a set of parameters that yield the desired transfer function) *without* the `mininput` and `minnoise` instructions. Because these instructions modify χ^2, they may upset the fitting algorithm so much that it fails to find a solution at all, if the starting values are too far off.
- For the `mininput` instruction, determine the desired input signal level as desired output level divided by the transfer function in the frequency range of interest. For example, for a 11 kHz bandpass filter with 10 dB gain and a desired output swing of ±12 V, an instruction like

[8] In models with ideal data and a final $\chi^2 \approx 0$, these indicators not always reliably indicate redundant parameters. In case of doubt disable one or more parameters and see if the same low χ^2 is still obtained. Any variation in χ^2 which is $\ll 1$ can usually be considered insignificant.

`mininput 3.79 10k 12k`
will do.

- Note that the dynamic range optimization will often reduce resistors down to the limit given by the op-amps output capability, leaving no room for an output load. This can be avoided by including the anticipated load as load resistor in the model.
- If in a `mininput` optimization the starting values provide the correct transfer function, but a far too low allowable input signal, it may be necessary to 'approach' a solution in smaller steps (i.e. first try to find a solution with a limit less strict than the final requirement, and then restart from that intermediate solution).
- The `minnoise` instruction can be controlled by the value of the '*aim*'. Good results have been achieved with an *aim* of around half the finally reachable noise. The reachable noise is usually not known in the beginning. An estimate can be found by running LISO with the `minnoise` instruction enabled, but using a huge value of *aim*, say 1 Volt. This effectively disables the `minnoise` instruction, but still prints the reached noise.
- For example, for the above-mentioned 11 kHz bandpass, a suitable instruction would be
  `minnoise no1 10k 12k 1.5u`
  if `no1` is the output node and the finally reachable noise is around $3\,\mu V_{rms}$.
- The weighting options also influence the result of a `minnoise` optimization. The propsed *aim* of half the reachable noise will yield a minimal χ^2 of around 1.4, which should correspond to only a negligible deviation of the transfer function from its goal. If that is impractical, choose another *aim* according to equation 6 on page 33.
- In the present implementation, `mininput` and `minnoise` both modify χ^2 in a similar way. The 'punishment' for a `minnoise` violation is applied exactly once, whereas the 'punishment' for a `mininput` violation is applied once for every data point in the frequency range given in the `mininput` instruction. Hence, if there are more than a few data points in the `mininput` frequency range, the `mininput` constraint becomes a much 'harder' limit than the `minnoise` constraint. If it is desired to make both of them equally 'hard', choose the `mininput` frequency range such that there is only one data point (*Untested*).
- `mininput` and `minnoise` both modify the χ^2 function in such a way that the Marquardt algorithm often becomes useless. If that happens, some computation time can be saved by invoking LISO with the `-s` option, ('use only Simplex').
- For the same reason, the correlation matrix and the estimated errors of the best parameters are usually wrong and useless. If they are needed, re-run LISO with `mininput` and `minnoise` disabled. It may be necessary to reduce the number of variable parameters to prevent the algorithm from running away from the optimal solution.

- If everything goes well, the optimization will proceed to yield a circuit with the minimum possible noise which at the same time has the desired transfer function and allowable signal swing. Once such a solution is found, it can sometimes be improved by restarting the fit, using the optimized parameters as starting values and adjusting the *aim* in `minnoise`.
- Further improvements may be possible by making more (or even all) components variable. While this is not a good idea in 'usual' fits, it may work when `mininput` and `minnoise` are properly set up and good starting values are given. It may even become unnecessary to specify limits for the parameters, since the constraints automatically restrict the components (in particular resistors) to reasonable values.
- **Any feedback on the success or failure of these instructions is very much desired by the author (`ghh@mpq.mpg.de`).**

10 Plot options

10.1 Defining the frequency range

Somewhere in the input file there must be a `freq` instruction that specifies the frequency range for which all computations are to be performed. It has the following format:

`freq lin|log` *startfreq stopfreq steps*

There must be exactly one of the keywords `lin` or `log`, specifying whether a linear or logarithmic subdivision is desired. The starting frequency *startfreq* and stopping frequency *stopfreq* must be given in Hertz, and the abbreviations listed in Section 5.1 on page 8 may be used. The integer *steps* specifies how many subdivisions of the frequency range should be made. The number of frequencies computed is

steps+1 (see also the footnote on page 47).

If in this run of LISO a fit is performed, the `freq` instruction can be omitted. The starting frequency *startfreq* and stopping frequency *stopfreq* are then taken from the data file and a logarithmic subdivision of 400 steps is assumed. If the `freq` instruction is present, it has priority over the limits taken from the data file.

10.2 GNUPLOT terminals

The GNUPLOT program that LISO uses for plotting can handle various output devices called *terminals*. The most important ones are `x11` for the X-Window system, `svga` for DOS systems and Postscript. LISO allows its output plots to be plotted on any combination of the available GNUPLOT terminals, and to specify various GNUPLOT options in addition. This is handled by the initialization file `fil.ini` in conjunction with the `gnuterm` instruction. Various GNUPLOT terminal description may be entered in the initialization file and given a name (see Section 11 on the following page). A `gnuterm` instruction has the following format:

`gnuterm` *terminal-name [file-name]*

where *terminal-name* must be defined in the initialization file and the optional *file-name* specifies where the output is written. For some terminals (such as X-Windows) it makes no sense to specify a file name. If the filename is not given, but needed, it is constructed from the basename of the current input file and a suitable extension (which again is specified in the initialization file). If several `gnuterm` instructions appear in the input file, the same plot is repeated for each of the specified terminals, in the order given. In that case, one usually wants to see the plot on the monitor in addition to writing it to a file. The instruction for plotting on the monitor should come last in this case, because otherwise the plot on the monitor will be deleted immediately after it is finished, because GNUPLOT can only handle a single output terminal at a time. For example, if you want the same plot both as an EPS file and on the terminal screen, you should enter

```
gnuterm eps
gnuterm x11
```

Note for X-Windows users: It is possible to leave the plot window on the screen even after the program has finished by specifying the `-persist` option for the GNUPLOT command. This can be done automatically by changing the corresponding line in the the initalization file `fil.ini` (see Section 11) to something like

```
GNUCOMMAND "/usr/local/bin/gnuplot -persist"
```

11 Initialization file "fil.ini"

Several parameters that are expected to be changed only occasionally (or never) are read from the file `fil.ini` after LISO starts. Here is a sample file that illustrates the file format:

```
PSTEP 5               # For Display "Computing 0...100%" update display
                      # after how many %
DEP_SAME_EPS 1e-3     # Relative difference, when dependent parameters
                      # are considered same.
EQSMALL 1e-13         # "same" for floating point comparison (not used)
PARMIN 1e-20          # parameters equal to zero will be set to PARMIN.
SMALL 1e-9            # Consider zero if below SMALL (equation dump)
TEMPERATURE 25        # Temperature in Celsius (for Johnson noise)
PUNISH_BETA 0.1       # Start heavy penalty when limits are exceeded
                      # by PUNISH_BETA
DERIV_H 1e-3          # Stepwidth for numerical derivative
ENDTOL 1e-4           # Fitting global exit criterion
SIM_STARTLEN 0.4      # Sidelength in initial simplex (Nelder-Mead)
SIM_LENFACT 3.3       # Divide SIM_STARTLEN by SIM_LENFACT for each
                      # subsequent call of Nelder-Mead
SIM_STARTTOL 1e-1     # Initial exit criterion for Nelder-Mead
SIM_TOLFACT 10        # Divide SIM_STARTTOL by SIM_TOLFACT for each
                      # subsequent call of Nelder-Mead
SIM_TOLMIN 1e-6       # Minimum denomin. in Nelder-Mead exit criterion
MAR_LAMSTART 1e-2     # Initial Guess for lambda in Marquardt
```

```
MAR_LAMFACT 1.5      # Multiplicative stepwidth for lambda
                     # searching in Marquardt
MAR_LAMMIN 1e-8      # Minimum lambda for Marquardt
MAR_REPTOL 1e-3      # Marquardt exit criterion
MAR_QUICKOUT 1e-5    # Quit at once if Marquardt gets below this ssq
SPA_REORDER 10       # Recompute Sparse Matrix fill-in scheme after
                     # frequency has increased by a factor SPA_REORDER

GNUCOMMAND "/usr/local/bin/gnuplot" # location of GNUPLOT executable
OPLIBNAME "opamp.lib" # opamp library

# gnuplot x11 terminal
GNUTERM x11
set term x11
END

# gnuplot eps terminal black-white
GNUTERM eps
SUFFIX eps
set term post eps enhanced mono dashed "Helvetica" 17
set size ratio 0.7
END

# gnuplot eps terminal colour
GNUTERM ceps
SUFFIX ceps
set term post eps enhanced color solid "Helvetica" 17
set size ratio 0.7
END

# gnuplot ps terminal
GNUTERM ps
SUFFIX ps
set term post landscape enhanced monochrome dashed "Helvetica" 17
set size ratio 0.7
END

# gnuplot ps terminal colour
GNUTERM cps
SUFFIX cps
set term post landscape enhanced color solid "Helvetica" 17
set size ratio 0.7
set key below
END
```

Some of the parameters are mentioned or explained in other sections (mainly Section 14). Normally the user will not often need to change any parameters. There is no error checking for the parameters in `fil.ini`, and it is easy to crash the program by entering unreasonable parameters.

The description of a GNUPLOT terminal has the following format:

`GNUTERM` *terminal-name*
[`SUFFIX` *default-suffix*]
GNUPLOT command
[*GNUPLOT command*] ...
`END`

The *default-suffix* is appended to the basename of the input file, if no explicit file name is given in the `gnuterm` instruction.

12 Op-amp library

The op-amp library is usually called `opamp.lib`. The name can be changed in the initialization file `fil.ini`, if different versions exist. It is again an ASCII file with '`#`' as comment indicator. It consists of several sections, each describing one type of op-amp. Here is a sample section illustrating the file format:

```
name = lt1037
a0 = 5e6
gbw = 53M
un = 2.5n
uc = 2
in = 0.4p
ic = 120
delay = 1n
pole = 870k # fitted from PMI OP37 data sheet
zero = 1.67M
pole = 17.1M
umax = 12
imax = 20m
sr = 15M
```

The `name` instruction must be the first instruction in the section. This name is used to refer to the op-amp in the input file. An entry is ended by either the next `name` instruction or by the end of the file. The meaning of each parameter will be explained in the remainder of this section.

12.1 Open-loop voltage gain

The open-loop voltage gain is in the simplest case described by the DC-open loop gain A_0 and the gain-bandwidth product f_{gbw}. This corresponds to a one-pole model for the op-amp gain, where the gain is given by

$$H(f) = \frac{A_0}{1 + A_0 \frac{s}{f_{\mathrm{gbw}}}} \tag{7}$$

`a0` is the DC open loop gain A_0. The unit is V/V = 1. Some synonyms in data sheets are *large signal voltage gain* or *open loop voltage gain*, and data sheets often use the units dB, V/mV or V/μV.

`gbw` is the gain-bandwidth product f_{gbw}, sometimes also called unity gain frequency or transit frequency. It is defined as the frequency where the magnitude of the open-loop voltage gain reaches unity. The unit is Hz. Note that for op-amps with external compensation the gain-bandwidth product depends on the compensation capacitor.

`delay` is a delay-time which causes extra phase lag at high frequencies.

`pole` introduces an extra pole in the transfer function.

`zero` introduces an extra zero in the transfer function.

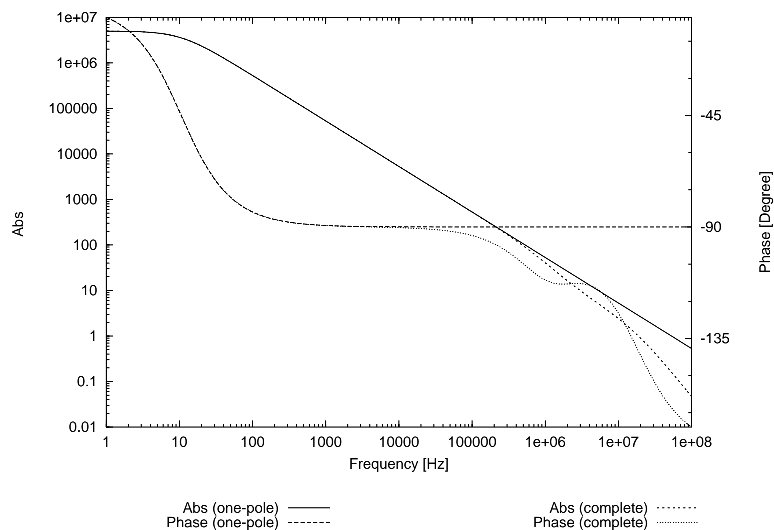
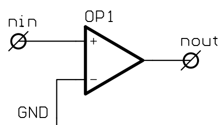


Figure 3: Open loop voltage gain of the op-amp described in the text, showing both the simple one-pole model and the complete model with two extra poles and one extra zero.

Figure 3 shows the open-loop gain that results from the one-pole model. It can be plotted (or fitted) using the following LISO model:




```
op op1 lt1037 nin gnd nout
```

```
freq log 1 100M 400
uinput nin 50
uoutput nout:abs:deg
```

This model can be refined by adding a delay time and further poles and zeroes to the transfer function. At present only simple poles and zeroes can be used. The basic gain as defined in equation (7) is then multiplied by $1 + \mathrm{i}\,(f/f_{\mathrm{zero}})$ for each zero and divided by $1 + \mathrm{i}\,(f/f_{\mathrm{pole}})$ for each pole. If the delay time is non-zero, the gain is multiplied by $\exp(-2\pi \mathrm{i} f\, t_{\mathrm{delay}})$. This causes an additional phase-lag. Note that extra poles and zeroes may change the actual unity-gain frequency. However, the gain-bandwidth product is not affected at frequencies which are well above f_{gbw}/A_0 but below the first pole or zero. Figure 3 on the page before also shows the open-loop gain that results from the complete LT1037 model[9] given above.

12.2 Op-amp noise

The following four parameters specify the op-amp noise:

`un` is the voltage noise spectral density in $\mathrm{V}/\sqrt{\mathrm{Hz}}$.

`in` is the current noise spectral density in $\mathrm{A}/\sqrt{\mathrm{Hz}}$.

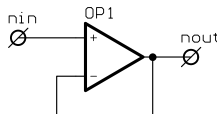
`uc` is the $1/f$ corner frequency of the voltage noise spectral density. The unit is Hz.

`ic` is the $1/f$ corner frequency of the current noise spectral density. The unit is Hz.

The voltage noise spectral density at a frequency f is computed as

$$\tilde{U} = \mathtt{un}\sqrt{1 + \frac{\mathtt{uc}}{f}} \tag{8}$$

and equivalently for the current noise. Note that the *voltage noise* spectral density at low frequencies (below `uc`) drops as $1/f^{1/2}$, whereas the *noise power* spectral density drops as $1/f$. Figure 4 on the following page shows as an example the voltage noise spectral density of the LT1037. It was produced using the following LISO model:




```
op op1 lt1037 nin nout nout
freq log 1 1k 400
uinput nin 50
noise nout op1:u
```

[9] Note that this is only given as an example, which does still not completely describe the open-loop gain on an LT1037 (it does, for example, not explain why a minimum gain of 5 is required for that op-amp). The reason is that the data from the data-sheet used to fit the poles and zeros extends only to just above than 10 MHz. With better data, the model could be improved.

To plot (or fit) the current noise of the noninverting input, the following model is useful:

```
op op1 lt1037 nin nout nout
freq log 1 1k 400
uinput nin 1
noise nout op1:+
```

Note that now the source impedance is important. By specifying it as $1\,\Omega$, a result corresponding to the equivalent input current noise measured in $\mathrm{A}/\sqrt{\mathrm{Hz}}$ is obtained.

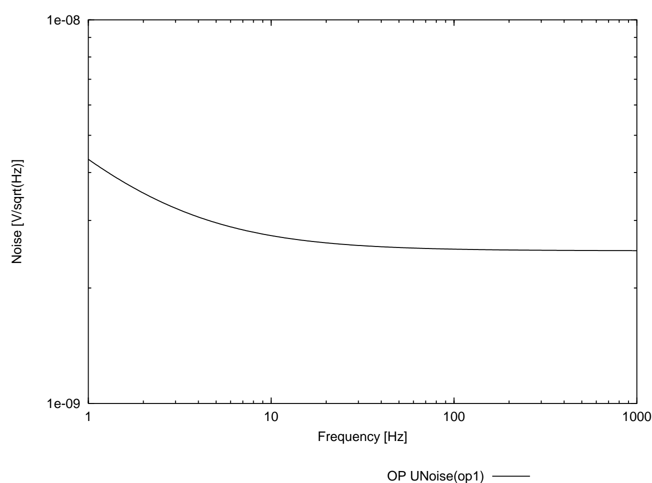


Figure 4: Voltage noise spectral density of the LT1037, computed from `un=2.5n` and `uc=2`
.

12.3 Op-amp limits

The three parameters `umax`,`imax`, and `sr` describe the maximum output voltage (in Volts), output current (in Ampère) and slew rate (in Volts/second) of the op-amp. These values are only used by the `maxinput` instruction (see section 7.1.5 on page 19).

13 Circuit simulation algorithm

13.1 Transfer functions

The fundamentals of the circuit simulation are surprisingly simple. They can be stated as follows:

First Kirchhoff law: The sum of all currents flowing into a node is zero.

Impedance of passive components: For a passive component with the (complex) impedance $Z(s)$, which is connected between *node1* and *node2* we have

$$Z(s)I = U_{node1} - U_{node2}, \tag{9}$$

where I is the current through that component flowing from *node1* to *node2*. For a resistor we have $Z(s) \equiv R$, for a capacitor $Z(s) = 1/(sC)$ and for an inductor $Z(s) = sL$, with $s = \mathrm{i}\omega = 2\pi \mathrm{i} f$.

Op-amp open loop gain: For an op-amp with open-loop gain $H(s)$, which is connected to the nodes 'p', 'm' and 'o' with its noninverting input, inverting input and output, respectively, we have

$$U_o = H(s)(U_p - U_m) \tag{10}$$

Note that for finite U_o and $|H(s)| \gg 1$, this reduces to $U_p \approx U_m$.

Voltage input: For a voltage input at the node *nin*, we set

$$U_{nin} = 1 \tag{11}$$

Current input: For a current we set

$$I_{in} = 1. \tag{12}$$

In the special case of a floating voltage input, the equation is

$$U_{nin} - U_{nin2} = 1, \tag{13}$$

where *nin* represents the input node and *nin2* the second input node of a floating voltage input.

Mutual inductances These are not so obvious. The procedure is as follows ([7, section 2.2.4]): If two coils L_1 and L_2 are coupled with the coupling factor k_{12} $(0 \leq k_{12} \leq 1)$, a mutual inductance M_{12} with the unit Henry is computed as

$$M_{12} = k_{12}\sqrt{L_1 L_2}. \tag{14}$$

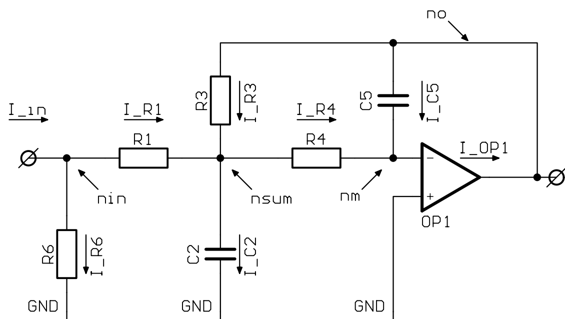
In the equation for L_1, which normally states something like

$$s\, L_1\, I_{\mathtt{L1}} - U_{\mathrm{node1}} + U_{\mathtt{node2}} = 0, \tag{15}$$

an additional term $+s\, M_{12}\, I_{\mathtt{L2}}$ is introduced. Correspondingly, in the equation for L_2, the term $+s\, M_{12}\, I_{\mathtt{L1}}$ is added.

We find that for each component we get one equation and one unknown (the current through that component). For each node, we also get one equation and one unknown (the voltage at that node). Furthermore, for the input we have one extra equation and also an extra unknown (the input current I_{in}). All these equations are linear in the

unknowns. This can most easily be demonstrated with an example, again using the low-pass filter of Figure 1 on page 13. We obtain the following equations[10]:



$$
\begin{aligned}
\frac{1}{s\,C_2} I_{\mathtt{C2}} - U_{\mathtt{nsum}} &= 0 \quad (\mathtt{C2}) \\
\frac{1}{s\,C_5} I_{\mathtt{C5}} - U_{\mathtt{no}} + U_{\mathtt{nm}} &= 0 \quad (\mathtt{C5}) \\
\frac{1}{H_{\mathtt{OP1}}(s)} U_{\mathtt{no}} + U_{\mathtt{nm}} &= 0 \quad (\mathtt{OP1}) \\
R_1 I_{\mathtt{R1}} - U_{\mathtt{nin}} + U_{\mathtt{nsum}} &= 0 \quad (\mathtt{R1}) \\
R_3 I_{\mathtt{R3}} + U_{\mathtt{nsum}} - U_{\mathtt{no}} &= 0 \quad (\mathtt{R3}) \\
R_4 I_{\mathtt{R4}} - U_{\mathtt{nsum}} + U_{\mathtt{nm}} &= 0 \quad (\mathtt{R4}) \\
R_6 I_{\mathtt{R6}} - U_{\mathtt{nin}} &= 0 \quad (\mathtt{R6}) \\
-I_{\mathtt{R1}} - I_{\mathtt{R6}} + I_{\mathrm{in}} &= 0 \quad (\mathtt{nin}) \\
-I_{\mathtt{C2}} + I_{\mathtt{R1}} + I_{\mathtt{R3}} - I_{\mathtt{R4}} &= 0 \quad (\mathtt{nsum}) \\
-I_{\mathtt{C5}} + I_{\mathtt{OP1}} - I_{\mathtt{R3}} &= 0 \quad (\mathtt{no}) \\
I_{\mathtt{C5}} + I_{\mathtt{R4}} &= 0 \quad (\mathtt{nm}) \\
U_{\mathtt{nin}} &= 1 \quad (\mathtt{input})
\end{aligned}
\tag{16}
$$

These are 12 linear equations for the 12 unknown variables
$I_{\mathtt{C2}}$, $I_{\mathtt{C5}}$, $I_{\mathtt{OP1}}$, $I_{\mathtt{R1}}$, $I_{\mathtt{R3}}$, $I_{\mathtt{R4}}$, $I_{\mathtt{R6}}$,
$U_{\mathtt{nin}}$, $U_{\mathtt{nsum}}$, $U_{\mathtt{no}}$, $U_{\mathtt{nm}}$ and I_{in}.

Note that for a current input, the first 11 equations remain unchanged and the last one is replaced by $I_{in} = 1$. This basic set of equations is already sufficient to compute the currents through all components and the voltages at all nodes. All results listed as "transfer functions" in section 7.1 on page 17 can be obtained from this set of equations.

[10] In the present version of LISO, these equations (with numerical values substituted) can be printed by the program for debugging purposes. To do so, disable any fitting instructions and enter a `freq` instruction like

```
freq lin 1k 1k 0
```

i.e. enter 0 'steps' to request the debugging output, and set both start- and stop-frequency to the frequency of interest.

13.2 Noise calculations

It turns out that a set of linear equations very similar to the one above can be used to compute the noise, if we now interpret all voltages and currents as linear spectral densities. We introduce noise sources into the circuit, as described below, and compute the voltage at a certain node, interpreted as linear spectral noise density. All component characteristics and Kirchhoff's law remain valid. We must, however, take care to compute only one noise source at a time, and then quadratically add the results. Otherwise different noise sources might be added with a complex phase, or even subtracted, whereas we want to treat all noise sources as independent and uncorrelated. Furthermore we only use the absolute value of the computed noise voltage, and not its phase, and hence the sign with which we introduce the noise source becomes unimportant.

13.2.1 Treatment of the input

As mentioned above, the input is treated differently for noise calculations. The last equation is replaced by

$$I_{in} Z_{in} + U_{input-node} = 0 \tag{17}$$

for the case of normal inputs and

$$I_{in} Z_{in} + U_{input-node} - U_{input-node2} = 0 \tag{18}$$

for a floating voltage input. This corresponds to the impedance Z_{in} connected between the input nodes.

13.2.2 Resistor noise

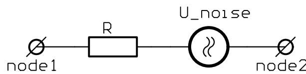
The equation describing a resistor,

$$R\,I - U_{node1} + U_{node2} = 0, \tag{19}$$

is replaced by

$$R\,I - U_{node1} + U_{node2} = U_n, \tag{20}$$

where U_n is the Johnson noise expressed as voltage noise source, $U_n = \sqrt{4kTR}$. This corresponds to connecting the noise source U_n in series with the resistor:



Note that as compared with the equations above, the *right hand side* is now changed, we still have the same unknowns and left-hand sides, but different *right hand sides.* The desired result is the voltage at a certain node, interpreted as linear spectral noise density.

13.2.3 Op-amp voltage noise

To compute the contribution of the voltage noise of an op-amp, we replace the equation describing the op-amp

$$U_o = H(s)(U_p - U_m) \tag{21}$$

by

$$U_o = H(s)(U_p - U_m + U_n). \tag{22}$$

This corresponds to amplifying not only the differential input voltage, but also the additional (equivalent input noise) voltage U_n (see Figure 5).

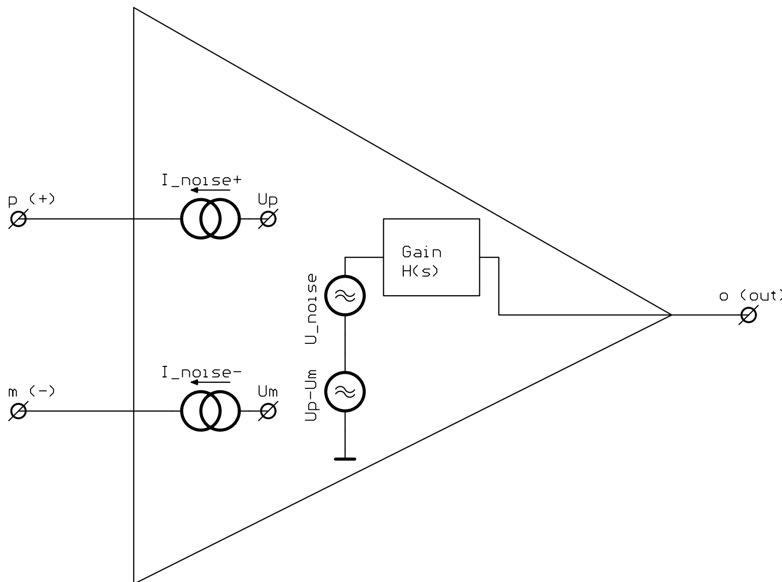


Figure 5: Noise model of an op-amp.

Note that there is no direct action of the voltage noise source to the input nodes. Without negative feedback (and non-zero op-amp gain), no contribution of the voltage noise appears at the input nodes. By rearranging the equation we get

$$\frac{1}{H(s)}U_o - U_p + U_m = U_n, \tag{23}$$

in other words, again only a change in the right-hand side.

13.2.4 Op-amp current noise

The current noise is modelled as a current source connected between the node where the op-amp input is usually connected, and its true, ideal input. Since the latter has

infinite impedance, no current can flow into it. Consequently the noise current must flow into the node where the input is connected. In other words, we inject the noise current into that node (see Figure 5 on the page before).

Assuming the previous equation describing that node was

$$I_1 + I_2 - I_3 - I_4 = 0, \tag{24}$$

we replace it by

$$I_1 + I_2 - I_3 - I_4 = I_n, \tag{25}$$

with I_n the noise current spectral density. Again this changes only the right-hand side of our equations.

13.3 Op-amp stability calculations

As already briefly mentioned in Section 7.2 on page 21, the following procedure is used to compute an op-amp stability function (see Figure 6):

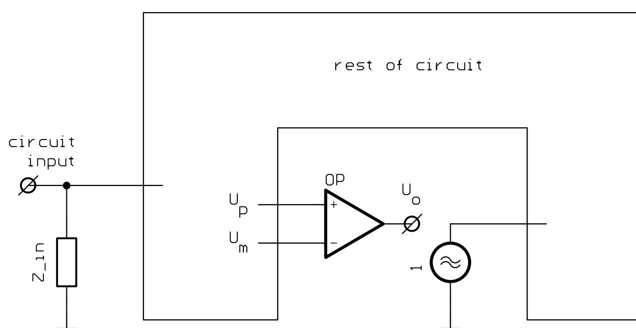


Figure 6: How the stability function of an op-amp is computed.

- The input is treated as described in Section 13.2.1 on page 48 (i.e. there is no input signal, and the specified source impedance is connected between the input nodes).
- The op-amp in question is "disconnected" from the circuit[11] and the voltage at its former output node is set to unity. Specifically, the equation (10) describing the op-amp is replaced by

$$U_o = 1. \tag{26}$$

[11] At present, the op-amps's output current is still included as a variable and added to the node where the op-amp's output used to be connected. It remains to be seen if this does any harm.

- The differential input voltage $U_p - U_m$ is computed. This represents the transfer function of the rest of the circuit from the op-amp's output to its input.
- That transfer function is multiplied by the op-amp's open-loop gain.

The result is the open-loop gain of the op-amp's feedback loop, which could theoretically be measured in the circuit by disconnecting the op-amp's output, injecting a test signal to where the output was formerly connected and measuring the transfer function from that test signal to the op-amp's output.

14 Fitting algorithms

The fitting algorithm is a combination of two "classical" algorithms (with some extensions), which are repeatedly looped over with tightening tolerances. The first one is the Nelder-Mead Simplex algorithm [1], a very robust geometrical algorithm which is good at approaching a minimum from far-off starting values. Secondly there is the Levenberg-Marquardt algorithm which is much faster once the vicinty of the (or a) minimum has been found. The author has learned about these algorithms and their proper application from his father Günther Heinzel, who has been using such algorithms for pharmacocinetic applications for many years.

14.1 Extensions to the Nelder-Mead Simplex algorithm

The algorithm is well described in the original paper[1]. It is implemented in LISO as described there. The only extension is the computation of start vectors:

14.1.1 Computation of $n+1$ directions equally distributed in R^n

In order to start the algorithm to minimize a function of n parameters, we need $n+1$ points in R^n. However, in our application only one initial guess is given. The same is true when the algorithm is restarted after apparent convergence to one point, which is urgently recommended because the simplex possibly has degenerated and no longer spans the whole parameter space.

Here a set of $(n+1)$ unity vectors $v_0, \ldots, v_n$, distributed as regularily as possible in R^n is useful. The problem has a unique solution (except for rotations) and the coordinates of the resulting points can explicitly be calculated by

$$
\begin{aligned}
v_0 &= (1, 0, \ldots, 0) \\
v_1 &= (b_0, a_1, 0, \ldots, 0) \\
v_2 &= (b_0, b_1, a_2, 0, \ldots, 0) \\
&\ldots \\
v_i &= (b_0, \ldots, b_{i-1}, a_i, 0, \ldots, 0) \\
v_{n-1} &= (b_0, \ldots, b_{n-2}, a_{n-1}) \\
v_n &= (b_0, \ldots, b_{n-2}, b_{n-1})
\end{aligned}
\tag{27}
$$

with the diagonal elements

$$
a_j = \sqrt{\frac{(n+1)(n-j)}{n(n-j+1)}}; \quad (a_0 = 1) \tag{28}
$$

and the subdiagonal elements

$$
b_j = -\sqrt{\frac{(n+1)}{n(n-j+1)(n-j)}} = \frac{a_j}{j-n}; \quad (b_0 = -\frac{1}{n}) \tag{29}
$$

Any two different vectors of this set have the scalar product

$$
v_i \cdot v_k = \cos \angle(v_i, v_k) = -\frac{1}{n} \tag{30}
$$

and the distance $|v_i - v_k| = \sqrt{2 + 2/n}$

14.1.2 Derivation

The basic observation is that $(n+1)$ equally distributed points cannot be given directly in R^n, but easily in R^{n+1}: $(1, 0, 0, \ldots, 0), (0, 1, 0, \ldots, 0), \ldots, (0, 0, \ldots, 0, 1)$. They all lie in the n-dimensional affine subspace $\{x = (x_0, \ldots, x_n) \in R^{n+1} \mid x_0 + x_1 + \cdots + x_n = 1\}$. By subtracting $(1/(n+1), 1/(n+1), \ldots, 1/(n+1))$ and normalizing, new unity vectors are obtained, now all in the linear subspace $H = \{x \in R^{n+1} \mid x_0 + x_1 + \cdots + x_n = 0\}$. Now the scalar product given above can readily be calculated, and the coordinates can be found either by constructing an orthonormal base of H and transforming the points' coordinates into this new base, or by computing the Cholesky–decomposition of the $(n+1) \times (n+1)$–matrix

$$
S = \begin{pmatrix}
1 & -1/n & -1/n & \ldots \\
-1/n & 1 & -1/n & \ldots \\
\ldots & & \ddots & \\
-1/n & \ldots & -1/n & 1
\end{pmatrix} \tag{31}
$$

14.1.3 Example

For $n = 3$ we get the 4 corners of a tetrahedron with center 0 and all points on the unity sphere:

$$\begin{array}{lll} (\ 1, & 0, & 0\ ) \\ (-\frac{1}{3}, & \frac{2}{3}\sqrt{2}, & 0\ ) \\ (-\frac{1}{3}, & -\frac{1}{3}\sqrt{2}, & \sqrt{\frac{2}{3}}\ ) \\ (-\frac{1}{3}, & -\frac{1}{3}\sqrt{2}, & -\sqrt{\frac{2}{3}}) \end{array} \tag{32}$$

14.1.4 Initialization and termination

Whenever the Simplex algorithm is called, it re-initializes the simplex by adding a fraction of the above unity vectors to the given inital estimate. The fraction (called starting length) is controlled by the overall minimization strategy (see Section 14.3 on page 56). In order to span all directions in parameter space as equally as possible, the unity vectors are randomly shuffled before each call of the Simplex algorithm. The algorithm then proceeds as usual. It is terminated as soon as the relative difference in χ^2 between the best and worst points in the present simplex falls below a certain limit (called tolerance), which is also controlled by the overall minimization strategy.

Note that the initial simplex does not contain the single point that was given as previous best estimate. Consequently in the first few iterations of the Simplex algorithm, the χ^2 may be worse than at the previous best point. This effect may look disturbing, but in my experience this procedure is more likely to escape from local minima in complex problems. If the previous best estimate was already the global minimum, the simplex just shrinks more or less symmetrically to that point and eventually reproduces the previous best χ^2.

14.2 Extensions to the Levenberg-Marquardt algorithm

14.2.1 Numerical derivatives

Since the Levenberg-Marquardt algorithm requires the derivatives of χ^2 with respect to the parameters, and χ^2, which results from the simulation, cannot be differentiated analytically, we use the following approximations for numerical derivatives:

$$\frac{\partial f}{\partial x} \approx \frac{f(x+h) - f(x-h)}{2h}, \tag{33}$$

$$\frac{\partial^2 f}{\partial x^2} \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}, \tag{34}$$

$$\frac{\partial^2 f}{\partial x \partial y} \approx \frac{f(x+h, y+h) - f(x+h, y-h) - f(x-h, y+h) + f(x-h, y-h)}{4h^2}. \tag{35}$$

The problem of selecting a suitable stepwidth h is simplified by the fact that the minimization takes place with the logarithms of all parameters, such that a fixed stepwidth h is usually appropriate. In the present implementation this h is a constant named `DERIV_H` read from `fil.ini` with the default value of 10^{-3}, corresponding to a 0.1% variation of the physical parameter. Note that any inaccuracy in the derivatives may affect the speed of convergence of the Levenberg-Marquardt algorithm, but not the final result.

14.2.2 Application to complex data

In a standard implementation of the Levenberg-Marquardt algorithm (see e.g. [2]), the gradient and Hessian of χ^2 with respect to the parameters are expressed by

$$\begin{aligned} \beta_k &= -\frac{1}{2}\frac{\partial \chi^2}{\partial p_k}, \\ \alpha_{kl} &= \frac{1}{2}\frac{\partial^2 \chi^2}{\partial p_k \partial p_l}, \end{aligned} \tag{36}$$

where p_k is the k-th parameter and χ^2 is defined in equation (5). For brevity we define

$$\begin{aligned} z_i &= y_i - \tilde{y}(f_i, p_1, \dots, p_m), \\ z_i^{k+} &= y_i - \tilde{y}(f_i, p_1, \dots, p_k + h, \dots, p_m), \\ z_i^{k-} &= y_i - \tilde{y}(f_i, p_1, \dots, p_k - h, \dots, p_m) \end{aligned} \tag{37}$$

and thus have

$$\chi^2 = \sum_i w_i \left|z_i\right|^2. \tag{38}$$

Note that in our case z_i is complex, different from standard implementations of the Levenberg-Marquardt algorithm. Using the approximation (33) we thus find for the first derivatives

$$\frac{\partial \chi^2}{\partial p_k} = \sum_i w_i \frac{\partial}{\partial p_k}\left|z_i\right|^2 = \sum_i w_i \frac{\left|z_i^{k+}\right|^2 - \left|z_i^{k-}\right|^2}{2h}, \tag{39}$$

$$\beta_k = \sum_i w_i \frac{\left|z_i^{k-}\right|^2 - \left|z_i^{k+}\right|^2}{4h}. \tag{40}$$

For the second derivatives we find

$$\begin{aligned} \frac{\partial^2 \chi^2}{\partial p_k \partial p_l} &= \sum_i w_i \frac{\partial^2}{\partial p_k \partial p_l}\left|z_i\right|^2 \\ &= \sum_i w_i \frac{\partial^2}{\partial p_k \partial p_l}(z_i\, z_i^*) \\ &= \sum_i w_i \left(\frac{\partial z_i^*}{\partial p_k}\frac{\partial z_i}{\partial p_l} + \frac{\partial z_i}{\partial p_k}\frac{\partial z_i^*}{\partial p_l} + z_i^* \frac{\partial^2 z_i}{\partial p_k \partial p_l} + z_i \frac{\partial^2 z_i^*}{\partial p_k \partial p_l} \right) \end{aligned} \tag{41}$$

At this point the last two terms in the parenthesis are dropped with the following reasons: The computational efficiency increases (we need only n_{param} evaluations of equation (33) instead of n_{param}^2 evaluations of equation (35)) and empirical evidence [2, 3] indicates that dropping these terms yields equally good results or even improved numerical stability. Furthermore, z_i should be small near the minimum of χ^2 and have a zero average. Finally, errors in α_{kl} and β_k affect only the speed of convergence, but not the final result. Now we apply equation (33) and obtain

$$\begin{aligned}
\frac{\partial^2\chi^2}{\partial p_k \partial p_l} &\approx \sum_i \frac{w_i}{4h^2}\left[\left((z_i^{k+})^* - (z_i^{k-})^*\right)\left(z_i^{l+} - z_i^{l-}\right) + \right. \\
&\qquad\qquad \left.\left(z_i^{k+} - z_i^{k-}\right)\left((z_i^{l+})^* - (z_i^{l-})^*\right)\right] \\
&= \sum_i \frac{w_i}{2h^2}\,\Re\left[\left((z_i^{k+})^* - (z_i^{k-})^*\right)\left(z_i^{l+} - z_i^{l-}\right)\right] \\
&= \sum_i \frac{w_i}{2h^2}\left(\Re[z_i^{k+} - z_i^{k-}]\,\Re[z_i^{l+} - z_i^{l-}] + \Im[z_i^{k+} - z_i^{k-}]\,\Im[z_i^{l+} - z_i^{l-}]\right), \\
\alpha_{kl} &= \sum_i \frac{w_i}{4h^2}\left(\Re[z_i^{k+} - z_i^{k-}]\,\Re[z_i^{l+} - z_i^{l-}] + \Im[z_i^{k+} - z_i^{k-}]\,\Im[z_i^{l+} - z_i^{l-}]\right).
\end{aligned} \tag{42}$$

For $k = l$ the last expression simplifies to

$$\alpha_{kk} = \sum_i \frac{w_i}{4h^2}\left|z_i^{k+} - z_i^{k-}\right|^2. \tag{43}$$

The function `derivs2` computes β_k and α_{kl} from equations (40) and (42). For debugging purposes, there is another function `truederivs` that uses the definining equations (5) and (36) together with (33)–(35) to compute the same quantities.

14.2.3 λ search

Now that we have found $\mathbf{A} = (\alpha_{kl})$ and $\mathbf{b} = (\beta_k)$ the algorithm proceeds as usual. We construct the augmented matrix

$$\begin{aligned}
\mathbf{A}'(\lambda) &= \left(\alpha'_{kl}(\lambda)\right) \quad \text{with} \\
\alpha'_{kk}(\lambda) &= (1+\lambda)\alpha_{kk}, \\
\alpha'_{kl}(\lambda) &= \alpha_{kl} \quad (k \neq l),
\end{aligned} \tag{44}$$

solve the set of linear equations

$$\mathbf{A}'\mathbf{d} = \mathbf{b} \tag{45}$$

for the corrections $\mathbf{d}(\lambda) = (\delta_k(\lambda))$, add the corrections δ_k to the parameters p_k and recompute $\chi^2(\lambda)$. Because the computation of $\mathbf{A}$ is comparatively expensive, our strategy to control λ differs from the standard procedure. After each computation of $\mathbf{A}$, we scan λ in steps of 1.5 (`MAR_LAMFACT`), starting with $\lambda = 10^{-3}$ (`MAR_LAMSTART`), with

limits 10^{-8} (MAR_LAMMIN) and 1000 (MAR_LAMMAX), until we have found the value of λ that minimizes $\chi^2(\lambda)$. Then we recompute $\mathbf{A}$ and $\mathbf{b}$ and start a new scan of λ. This procedure is repeated until the relative improvement in χ^2 becomes less than 10^{-3} (MAR_REPTOL).

Note that if constraints are violated, the χ^2 function used in the λ search is increased due to the 'punishment function' (see 14.4), whereas $\mathbf{A}$ and $\mathbf{b}$ are unaffected by the constraints. In this situation the Marquardt algorithm will usually fail and return with no improvement, leaving the work to the Simplex algorithm.

14.3 Overall minimization strategy

At present the following strategy is implemented: The main loop first calls the Simplex algorithm with a tolerance of 10^{-2} (SIM_STARTTOL) and a length factor of 0.4 (SIM_STARTLEN). Then the Marquardt algorithm is called, after which it loops back to the Simplex algorithm. Each time the Simplex algorithm is called, the length factor is divided by 3.3 (SIM_LENFACT) and the tolerance by 10 (SIM_TOLFACT). The optimization is always ended after a Simplex step, such that normally the minimum course of events is Simplex–Marquardt–Simplex. The criterion for ending is a relative improvement in χ^2 below 10^{-4} (ENDTOL) in the last Simplex step. There is, however, a shortcut for very good fits (which usually occur with simulated but almost never with measured data): If after the Marquardt algorithm χ^2 is below 10^{-5} (MAR_QUICKOUT), the iteration is stopped immediately. This feature can be disabled by setting (MAR_QUICKOUT) to a very small number (e.g. 10^{-13} or so) in the file `fil.ini`.

Because the parameters may differ widely in magnitude (e.g. pF vs. MΩ), all optimization algorithms work with the logarithms of the parameters. A consequence is that only positive parameters are allowed. The transformed parameters now all have a range of approximately $-20\ldots20$ or a subset thereof, with a precision of 0.001 being adequate for most purposes. This transformation is essential for the efficiency and reliability of both the Simplex and the Marquardt algorithm, which have some geometrical background.

14.4 Implementation of parameter limits

There are at least three possibilities to implement limits for the parameters:

- With a transformation such as a suitably scaled $\arctan(x)$ function new parameters can be introduced which may vary between $-\infty$ and ∞. The possible huge values of the transformed parameters might, however, upset geometrical algorithms.
- Any proposed set of new parameters can be examined for crossing the limits, and then approporiate action may be taken, such as clipping at the limits. This is a possible approach, but rather complicated to implement into already complex algorithms.

- The function to be minimzed, χ^2, may be modified such that a "penalty" is added whenever the limits are crossed.

Since this last possibility (penalty function) is the most flexible[12] one, is easy to implement and has proven very satisfactory so far, it was chosen for LISO.

The penalty function was chosen as follows: Define

$$f(x) = \begin{cases} \exp(x) - x - x^2/2; & x > 0 \\ 1; & x \leq 0 \end{cases} \tag{46}$$

with $f(x) \equiv 1$ for $x < 0$, $f(1.57) \approx 2$, $f(x)$ increasing rapidly for $x > 0$ and twice differentiable at $x = 0$. Now consider the parameter p_i with limits $\min_i$ and $\max_i$. Here parameters and limits are taken directly with their physical values (i.e. neither scaled nor as logarithms). If p_i falls outside $\min_i$ or $\max_i$, compute $x > 0$ as follows:

$$x = \begin{cases} (p_i/\max_i) - 1; & p_i \geq \max_i \\ (\min_i/p_i) - 1; & p_i \leq \min_i \end{cases} \tag{47}$$

Then multiply x by $1.57/\alpha$, where α has been precomputed as

$$\alpha = \begin{cases} \beta; & \max_i / \min_i > 2 \\ \beta(\max_i/\min_i) - 1; & \max_i / \min_i \leq 2 \end{cases} \tag{48}$$

with the constant $\beta = 0.1$ (`PUNISH_BETA` in the initialization file).

The function χ^2 is then multiplied with $f(x)$ for all parameters which exceed their limits. The effect is that χ^2 is doubled if p_i has passed beyond a limit by 10% (of that limit), with the "10%" being reduced if $\min_i$ and $\max_i$ are close together.

A related strategy (*still experimental*) is used for the `mininput` instruction. For all data points with $f_{\min} \leq f \leq f_{\max}$, the allowable input signal S_a is computed (in the same way as for the `maxinput` instruction) and compared to the required signal S_r (which was given by the user in the `mininput` instruction). If $S_a < S_r$, the following sum is incremented:

$$a = \sum_{S_a < S_r} \log \frac{S_r}{S_a}. \tag{49}$$

After looping over all data points, a has a value between 0 (for no constraint violation) and a few thousand (for heavy constraint violation in many data points). This value a is then transformed by

$$b = 20 \arctan(a/10), \tag{50}$$

which is linear for small a, monotonuously increasing and limited to 10π. The previous χ^2 is then multiplied by $\exp(b)$. If the computed transfer function agrees perfectly with the data file, χ^2 is very close to zero and the multiplication by $\exp(b)$ was found to provide not enough 'punishment' in some cases. Hence the modified χ^2 is set to

$$\chi^2_{\text{mod}} = \min(\chi^2 \exp(b), b). \tag{51}$$

[12] It is, for example, possible to implement more complex constraints very easily.

The χ^2 thus modified is used for the Simplex algorithm and the λ search of the Marquardt algorithm. The gradient and Hessian for the Marquardt algorithm (equation 36 on page 54) as well as the variance–covariance matrix (see Section 14.5) are *not* affected by the limits. This means that if one or more parameters are constantly running against a limit, the Marquardt algorithm may be not working and convergence to a minimum may then rely only on the Simplex algorithm.

If that final minimum of the modified χ^2 involves a parameter having crossed its limit, the correlation matrix and confidence limits (see Section 14.5) may be wrong, because we are not at a minimum of the unmodified χ^2.

14.5 Correlation matrix and confidence limits

After the fit has converged, the matrix $\mathbf{A} = (\alpha_{kl})$ is computed according to equation 42 on page 55 and inverted to give the variance–covariance matrix

$$\mathbf{C} = (c_{kl}) = \mathbf{A}^{-1}. \tag{52}$$

The correlation between two parameters p_k and p_l is now computed as

$$\mathrm{corr}_{kl} = \frac{c_{kl}}{\sqrt{c_{kk}\, c_{ll}}}. \tag{53}$$

The standard deviation σ_k of p_k is computed as

$$\sigma_k = \sqrt{\frac{\chi^2\, c_{kk}}{n_{\mathrm{data}} - n_{\mathrm{param}}}}. \tag{54}$$

This standard deviation refers to the logarithmic parameter and is transformed back into physical parameters by

$$re = \exp(\sigma_k) - 1 \tag{55}$$

where re now means the relative error in the parameter. Note that for $\sigma_i \geq \log 2 \approx 0.69$ the relative error re becomes equal or bigger to the best estimate of the parameter because

$$\log(p_{\mathrm{best}}) - \sigma_i \leq \log p_i \leq \log(p_{\mathrm{best}}) + \sigma_i, \tag{56}$$

which at the limit $\sigma_i = \log 2$ transforms into

$$0.5\, p_{\mathrm{best}} \leq p_i \leq 2\, p_{\mathrm{best}}. \tag{57}$$

"Standard deviations" such big (or bigger) are only of very limited usefulness anyway. Note also that both correlation matrix and confidence limits may be wrong, if the final minimum of χ^2 involves any parameter having crossed its limit (see Section 14.4 on page 56).

15 Implementation notes

15.1 Internal scaling

For most numerical algorithms, it is advantageous or even necessary to *scale* the variables, i.e. to express them in some kind of natural unit instead of their SI units. This is in particular true in LISO, where we deal with e.g. pF and MΩ, nA and MHz. The purpose is to get internal variables that have at least comparable orders of magnitude. Without such scaling, many algorithms may become imprecise, inefficient or may even fail. In LISO, the following internal scaling is used for impedances, capacitances, inductances, voltages, currents and frequencies:

$$R = R' \, R_n \tag{58}$$
$$C = C' \, C_n \tag{59}$$
$$L = L' \, L_n \tag{60}$$
$$U = U' \, U_n \tag{61}$$
$$I = I' \, I_n \tag{62}$$
$$f = f' \, f_n. \tag{63}$$

Here R (as an example) stands for the real physical values of the impedance (with the unit Ω), R' is the internal value used by LISO algorithms (without any unit), and R_n represents the scale factor (again with the unit Ω). Out of various possibilities to select reasonable scale factors, LISO uses the following one:

$$R_n = R_{\text{mean}} \tag{64}$$
$$C_n = C_{\text{mean}} \tag{65}$$
$$L_n = R_n^2 \, C_n \tag{66}$$
$$U_n = 1\,\text{V} \tag{67}$$
$$I_n = U_n / R_n \tag{68}$$
$$f_n = 1/(R_n \, C_n), \tag{69}$$

where R_{mean} and C_{mean} are the geometric mean values of all resistors and capacitors in the circuit, respectively. Another possibility, which might even be better for computational purposes, but which would make the program more complex, would be to choose f_n equal to the frequency just being considered, and choose the other scale factors accordingly. For simplicity we have, however, chosen the above scale factors, which remain constant for any single run of LISO. The numerical values of the scale factors are printed at the end of the output file.

15.2 Sparse matrices

For each frequency, a system of linear equations (such as equation (16)) needs to be solved numerically. Even for circuits of moderate complexity, that system gets already rather large (its dimension is: number of components + number of nodes +1). Of

course it is possible to apply standard techniques, such as LU-decompositon. However, the computation time required makes that approach very impractical (in particular for the fitting function, which needs to solve the system many times). It turns out, however, that most elements of the matrix describing the system are zero. Such matrices are called *sparse*, and there are special techniques available to deal with them. LISO uses a package called SPARSE [4], which was developped for circuit simulation and is therefore very well suited for this application. Practically, on a "normal" computer, at least the fitting function of LISO would be unusably slow without such a specialized sparse matrix package.

15.3 Efficient noise computation using the transposed matrix

In one run of LISO, the noise is computed only at one fixed node. Usually, several or all noise contributions are to be computed simultaneously. If we write the linear system (16) in matrix form, we get

$$Ax = b \tag{70}$$

with A the matrix describing the left-hand sides of (16), b the right-hand sides and x the solution vector (consisting of all voltages and currents in the circuit). Formal inversion yields

$$x = A^{-1}b. \tag{71}$$

We are only interested in one component of x, say x_n, which we call y (a scalar):

$$y = x_n = (A^{-1}b)_n. \tag{72}$$

The index n represents the node of interest in the solution vector and is fixed. As explained in Section 13.2, every noise contribution corresponds to a different right-hand-side vector b, say

$$b_j = k_j e_j \tag{73}$$

with k_j some real constant and e_j the unity vector $(0, \ldots, 1, \ldots, 0)$ with the '1' at position j. So the jth noise contribution can be computed as

$$y_j = (A^{-1}b_j)_n = k_j(A^{-1}e_j)_n. \tag{74}$$

If more than one noise contribution is to be computed, we need to *separately* compute each of them and then add them quadratically:

$$\tilde{U} = \sqrt{\sum_j |y_j|^2}, \tag{75}$$

a nonlinear operation. (As explained in Section 13.2, we cannot directly compute different noise contributions in one operation). In a straightforward approach, we

would have to evaluate equation (71) separately for each b_j given by (73), an obviously inefficient procedure. We can, however, rewrite equation (74) as follows:

$$y_j = k_j \sum_l (A^{-1})_{nl}(e_j)_l = k_j (A^{-1})_{nj}. \qquad (76)$$

We see that all the different noise contributions are contained in one *row* (with index n) of the matrix A^{-1}. We can compute that row in one operation if we solve the equation

$$A^t \hat{y} = e_n, \qquad (77)$$

where A^t represents the transpose of A. The elements of $\hat{y}$ can be thought of as transfer functions from the various points where noise sources might reside to the specified node where the noise is to be computed. Then we can immediately compute

$$y_j = k_j \hat{y}_j \qquad (78)$$

and apply (75).

16 Application examples

In this Section, several simple examples are presented to help the reader understand the possibilities of LISO and the necessary input. They are not intended to exhaust all of LISO's possible uses. They are sorted by increasing complexity.

16.1 Root mode simulation

The aim is to compute the frequency response of a 4-pole Tschebyscheff filter with 3 dB ripple, a corner frequency of 4 kHz and a DC-gain of 10. From filter tables[6, 5], we obtain the following pole data:

Stage 1: $\Omega = 0.44265, \quad Q = 1.0765$
Stage 2: $\Omega = 0.9503, \quad Q = 5.5770$

Now all we need is a frequency scaling (from 1 Hz to 4 kHz) and a gain scaling (from 1 to 10). The input file `ex1.fil` is:

```
pole .44265 1.0765  # from filter tables
pole .9503 5.577    # from filter tables
ffactor 4000        # frequency factor
factor 10           # gain factor
tfoutput db:deg-
freq log 100 100k 400
gnuterm eps
gnuterm x11
```

Note the use of the `deg-` coordinate specifier (see Section 7.1.2 on page 18) in order to avoid an ugly phase jump at 180°. The resulting plot is shown in Figure 7 on the following page.

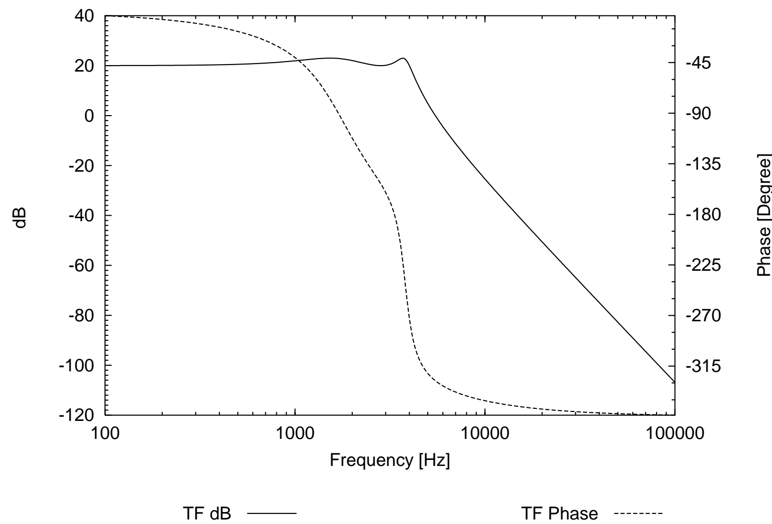


Figure 7: Example 1: Frequency response of a 4-pole Tschebyscheff filter.

16.2 Root mode fit

The transfer function of a suspended pendulum was measured with a spectrum analyzer. (These are real experimental data, the transfer function is from a current in a coil to the actual movement of the penulum, measured optically.) It has the general form of a two-pole low-pass filter:

$$H_\mathrm{P}(s = \mathrm{i}\,f) = \frac{A_\mathrm{P}}{1 + \dfrac{s}{f_\mathrm{P} Q_\mathrm{P}} + \dfrac{s^2}{f_\mathrm{P}^2}} \tag{79}$$

We want to find the pole frequency f_P, its associated Q_P and the overall gain factor A_P. The data from the spectrum analyzer was stored in the file called `ex2.dat`. It consists of 3 columns: first the frequency (in Hz), second the real part and third the imaginary part of the transfer function. The frequency runs from 0 to 100 Hz in 801 steps. The input file `ex2.fil` is:

```
pole 1.5 2      # pole at f=1.5 Hz with Q=2
factor -10      # overall factor is negative
freq log 0.1 100 400
tfoutput abs:deg
param pole0:f 0.1 10
param pole0:q 0.1 10
param factor 1 100 # note the factor used for fitting is positive!
fit ex2.dat reim rel plotfirst
gnuterm eps
gnuterm x11
```

Note that the overall factor is negative. Its starting value (defined by the instruction `factor -10`) must be negative in this case, because the fitting algorithm cannot "cross" from negative to positive values and vice versa. The limits in the `param` instruction as well as intermediate values in the printout are positive. The overall factor in root mode is the only parameter in LISO that can have negative values and has this peculiar behaviour. All other parameters are always positive. The instruction `fit ex2.dat reim rel plotfirst` gives the filename, file format and weighting. The option `plotfirst` requests a plot of the function with the startvalues as well as one with the final fitted values. Note also that the resulting plot (shown in Figure 8 on the next page) has a logarithmic x-axis and the coordinates `abs:deg`, although the data file has linearly spaced x-values and the coordinates `reim`. The first data point at the frequency 0 Hz cannot be plotted. Running LISO on this example yields the following output:

```
nfunk  Stat       ssq   pole0:f   pole0:q    factor
    1 Start      1924       1.5         2        10
    5  Splx     243.2     1.242     1.443     8.752
    7  Splx     32.69    800.1m         2     12.49
   17  Splx     27.13    990.7m      2.32     9.693
   21  Splx     24.39    883.3m     1.963     10.96
   34  Sout     22.28    898.4m     2.848     11.16
   34  M in     22.28    898.4m     2.848     11.16
   54  Marq     22.12    912.3m     2.954     10.69
   54  M in     22.12    912.3m     2.954     10.69
   64  Marq     22.12    913.5m     2.955     10.66
   64  Mout     22.12    913.5m     2.955     10.66
   68  Splx     35.06    913.5m     2.955     12.04
   70  Splx     25.62      949m     2.955     10.52
   78  Splx     22.72    900.2m     2.929     10.68
  100  Sout     22.14    910.4m     2.976     10.78

Correlation matrix
       pole0:f pole0:q  factor
pole0:f       1
pole0:q   0.169       1
 factor  -0.973  -0.183       1

Best parameter estimates:
pole0:f =  913.50261m +- 11.48m (1.26%)
pole0:q =  2.955091 +- 227.8m (7.71%)
factor =  -10.663409 +- -277.8m (2.6%)
```

Note that the final minimum of SSQ is already found by the first application of the Marquardt algorithm (after 64 function evaluations). The subsequent call of the Simplex algorithm cannot improve this minimum. The final results are always the best ones ever found during the whole fitting procedure. The printed "deviations" of the best parameter estimates have no real meaning, unless the data have normally distributed

errors and these errors are known and entered (which is not the case in this example). The correlation matrix is, however, still valid.

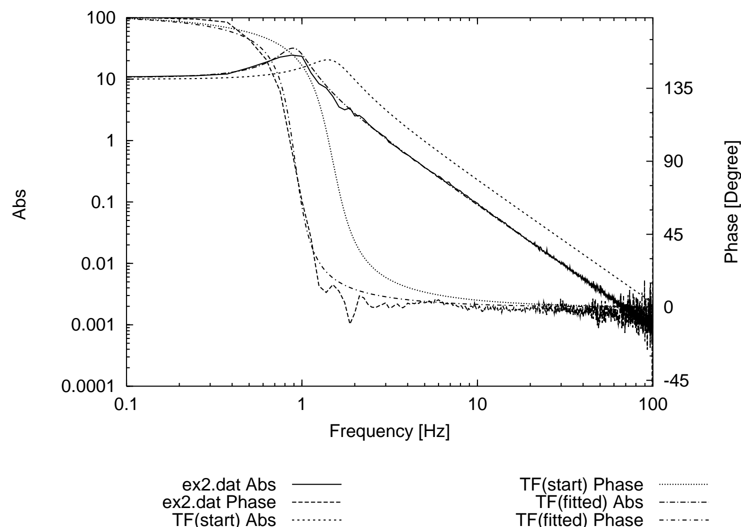


Figure 8: Example 2: Fitting a measured frequency response to a 2-pole lowpass filter model.

References

[1] J. A. Nelder, R. Mead: A simplex method for function minimization, Computer Journal **7** (1965), 308–313

[2] William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery: Numerical Recipes in C, The Art of Scientific Computing, Cambridge University Press, 2nd ed., 1992.

[3] J. R. Donaldson, R. B. Schnabel: Technometrics **29** (1987), 67–82

[4] *SPARSE*: A Sparse Linear Equation Solver, Version 1.3a, 1 April 1988, by Kenneth S. Kundert and Alberto Sangiovanni-Vincentelli, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, Calif. 94720; available from `http://www.netlib.org/sparse/`

[5] M. Herpy, J.–C. Berka: Aktive RC–Filter, Franzis–Verlag München 1984

[6] L. v. Wangenheim: Aktive Filter in RC– und SC–Technik, Hüthig–Verlag Heidelberg 1991

[7] E. E. E. Hoefer, H. Nielinger: SPICE – Analyseprogramm für elektronische Schaltungen, Springer–Verlag 1985

[8] S. Brandt: Datenanalyse, B.I. Wissenschaftsverlag 1992