ExpLab
A Tool Set for Computational Experiments

Susan Hert   Lutz Kettner
Tobias Polzin   Guido Schäfer

MPI–I–2002–1-004   December 2002

Authors' Addresses

Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken
[hert,polzin,kettner,schaefer]@mpi-sb.mpg.de

Acknowledgements

Partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

Abstract

We describe a set of tools that support the running, documentation, and evaluation of computational experiments. The tool set is designed not only to make computational experimentation easier but also to support good scientific practice by making results reproducable and more easily comparable to others' results by automatically documenting the experimental environment. The tools can be used separately or in concert and support all manner of experiments (*i.e.*, any executable can be an experiment). The tools capitalize on the rich functionality available in Python to provide extreme flexibility and ease of use, but one need know nothing of Python to use the tools.

Keywords

Computational Experiments, Algorithm Engineering, Performance Benchmarking

1 Introduction

This document describes a set of tools that **support the running, documentation, and evaluation of computational experiments**. We take our cues from the natural sciences where experiments are performed in a lab that is equipped with tools designed specifically for the purpose of supporting experimentation and where experiments are fully documented in a lab book. This is done so the results can be analyzed in the proper context and others can reproduce the same experiments to verify the results or perhaps test hypotheses about which parts of the experimental context are responsible for the observed results. The informational needs of scientists evaluating computational experiments are exactly the same as those for evaluating other types of experiments: the context must be known and, to lend credibility to the results, one must be able to reproduce the same experiment in one's own lab from the data presented. Such reproducibility is recognized, not only by scientists, as an important part of scientific practice [Joh96, MPG00], as is the ability to describe precisely the experimental environment when presenting results [MM99].

1.1 Goals

There are three main goals that motivate the development of this tool set:

- to provide a simple way to set up and run computational experiments;
- to provide a means of automatically documenting the environment in which an experiment is run so the experiment can be easily rerun (provided the same environment is still available) and the results can be more accurately compared to the results of other computational experiments;
- to eliminate some of the tedium involved in collecting and analyzing output by providing basic text output processing tools.

It must be noted that our goal here is not to replace existing tools that already provide useful functionality for computational experiments (*e.g.*, `gnuplot`, `make`, `perl`, `python`). Rather, the goal is to augment this set with new tools that build on the functionality already available to provide a comfortable experimentation environment.

1.2 Supported Platforms

We believe the tool set works on most Solaris, IRIX, and Linux platforms on which Python version 1.6 or greater is installed. We have tested it on

- Solaris 5.8 and 5.9,
- IRIX 6.5, and
- Linux 2.2.19 (Debian).

We may unknowingly support other platforms as well. If you use the tools successfully on some other platform, please let us know so we can extend this list.

1.3 Supported Experiments

Because we do not want to restrict the user by limiting the set of commands that can be used in running an experiment, our notion of an experiment is very general; it is, in fact, simply an executable. This means that a user may supply, for example, a compiled binary file or an executable shell script as the experiment. This notion of an experiment supports not only different computational contexts but also different programming styles.

An experiment generally consists of three separate stages: setup (Section 2), execution (Sections 3 and 4), and analysis of results (Section 5). Our tool set is designed with these three stages in mind but allows users to bypass any one of the stages so the tools could be used, for example, to help in analyzing existing output files. Each of the tools in our set has a command-line interface following the style of other common Unix tools (Section 7.1). Recognizing that it can be tedious and error-prone to provide a large (or even a small) set of command-line arguments that activate the features one wants to use, all the tools also allow input of some or all arguments from a text file (Section 6.2).

2 Setting up an Experiment

There are (at least) three things involved in setting up a computational experiment. The first, very important one, is establishing a revision control repository for the source files that are used to construct the executable (*i.e.*, the source code **and** the makefile(s)). This is necessary to assure that experiments are reproducible. Our tools are designed assuming CVS is used for this version control. It is possible to use the tools without source code revision control, but this is, of course, not what we recommend. The accompanying tutorial for our tools gives a brief introduction to the use of CVS for those not familiar with it.

The other two things to be set up are the instruments that record the relevant environmental context and the executable and its input that will serve as the experiment. Our tool set provides assistance for both of these setup tasks.

2.1 Recording the Experiment's Context

For recording environmental context, the tool `labsetup` is provided. It generates a resource file that contains information provided by the user in response to queries about certain commonly relevant environmental data. By default, this configuration file is called `.labrc` and is stored in the user's home directory. If such a file already exists, it will be modified according to the input provided and the old version of `.labrc` will be copied to `.labrc.bak`. Alternately, the user can indicate that a resource file, `labrc` should be created (or modified) in the current directory.

The tools use the information stored in such configuration files, if present, to augment the data provided via the command line. For options that may have a single value only, data in the local `labrc` file has precedence over data in the global file `($HOME)/.labrc`, and options given at the command line have precedence over options provided in a configuration file. For options, such as `--cvs` for `labrun`, that may appear more than once on a command line, the union of the option values from the command line and the resource files, if any, is used by the tool.

Here is an excerpt from a sample run of `labsetup` showing how you can specify that the C compiler and its version should always be logged as well as the name of the graphics card being used.

```
labsetup session
labsetup, Revision 1.13, 2002/07/09

For all questions asked, the current value represented by existing
~/.labrc is displayed in square brackets after the question.

If no value is set in ~/.labrc, the default value is shown
in brackets.  To keep the value shown, simply hit return.

To reset a flag not corresponding to a yes-no question to the default
use 'use default' as the input value.
----------------------------------------------------------------------

[...]

Change settings for labrun? [y]
y

[...]

Environment variables whose values should be noted
   Enter new values one per line.

   Use a '+' at the beginning of the first line to add to the current
   list; otherwise any new values given will replace the current ones.

   An empty line ends the input.
[None]
CC


Additional comments to be recorded in log file
   Enter new values one per line.

   Use a '+' at the beginning of the first line to add to the current
   list; otherwise any new values given will replace the current ones.

   An empty line ends the input.
[None]
Compiler version=`$CC -v`:version
Graphics card=`hinv`:Graphics\sboard
```

```
[...]
```

2.2 Specifying the Executable and Its Parameter Sets

Though our tool set is general enough to support a myriad of experiments, we imagine the most common experiments will be one of the following three kinds:

- compiling programs with different compilation flags and running them on the same input;
- running programs on different input sets;
- running one of the above two types of experiments on different machines.

We provide specific means to make these types of experiments easier.

2.2.1 The `labmex` Tool

The tool `labmex` (for "Make and EXecute") supports the first kind of experiment. As its name implies, this tool can be used to compile and execute a given program. By default, the command `make` is used, together with any user-supplied options for `make`, to build a certain target (or targets). If compilation succeeds, then the specified executable is run with any given command-line arguments. If compilation fails, a compilation log is generated with the name `<executable>-<date>-<time>.clog`. By default, this file will reside in the current directory, or, if called by `labrun` it will reside in the directory in which `labrun`'s log files are stored.

For example, the command

```
labmex fun 123
```

would translate into the commands

```
make fun
fun 123
```

If the `make fun` command fails, a file with a name like `fun-2002-05-09-205405.clog` would be created containing the output from the `make` command, and `fun 123` will obviously not be executed. In addition, a link to the compile log file with the name `current.clog` will be created in directory containing the log file, thus making it easier to access the latest compile log.

Other command-line options for this tool allow one to specify a different directory for the compilation log file, target or targets other than the executable for the compilation command, a command other than `make` for building the targets and that a `make clean` should be done before, after, or before and after the target is built.

The `labmex` tool can be used outside the context of computational experimentation as a general shortcut tool for compiling and executing any program.

2.2.2 Multiple Data Sets, Compilations, or Machines

Though conducting an experiment using multiple data sets or multiple sets of compilation flags can be easily accomplished by issuing multiple separate experiment commands, it is a bit tedious to do things this way. One can easily forget which data sets have been tested and which have not. We have therefore provided ways around (some of) the tedium and (some of) the forgetfulness. For the tools `labrun` (Section 3) and `labrerun` (Section 4), the keyword `NEX` (for "new experiment", or simply "next" if you prefer) can be used in the command line to indicate that what follows is the input for a new experiment. Thus multiple experiments can be specified at once. See Section 6.2 and the examples accompanying the description of `labrun` in Section 7.3 for examples.

Any number of `NEX`'s are possible. This feature is most useful when command-line arguments are provided via a file instead of on the command line itself (Section 6.2).

For running experiments on different machines, our tools can be used quite easily in conjunction with the `ssh` command. For example, one could compile and execute a program on a specific machine `turing` using the command

```
ssh turing labmex fun 123
```

See the documentation for `ssh` for more details on remote execution of commands. Using `ssh` in conjunction with the `NEX` keyword allows one to run an experiment on several different machines quite easily. See the examples accompanying the description of `labrun` in Section 7.3 for an example.

2.2.3 The `labschedule` Tool

For more elaborate scheduling of multiple experiments, the tool `labschedule` is provided. This tool provides an easy means to

- loop through sets of input values,
- start several experiments simultaneously on one machine,
- distribute a set of experiments among a cluster of machines.

This tool's extreme flexibility comes through the use of loops and variables. Variable names begin with a `%` and loops are designated using the `--for` option.

Each loop has an associated variable that is simply the number of the loop in the command line preceded by a `%`. For example, to run several experiments that differ only in the arguments given to the program, a single `for` loop will suffice.

```
labschedule --for='10 20 30' bench %1
```

This command will cause the three experiments

```
bench 10
bench 20
bench 30
```

to be started in succession on the local machine. More precisely, the following three `labrun` commands will be issued:

```
labrun --name=schedule-10 bench 10
labrun --name=schedule-20 bench 20
labrun --name=schedule-30 bench 30
```

The `--print` option of `labschedule` will show you the commands that are to be executed with all variable names replaced with the corresponding values. Any number of `--for` options are possible, and the result will be a nested loop structure, with the first `--for` command corresponding to the outermost loop. For example,

```
labschedule --for='10 20 30' --for='a b' bench %1 %2
```

will, in essence, cause the following to be executed:

```
for %1 in [10, 20, 30] do
   for %2 in [a, b] do
      labrun --name=schedule-%1-%2 bench %1 %2
```

and thus six experiments will be started in succession.

There is a rich syntax available for specifying the ranges of the `for` loop variables. One can use python expressions (*e.g.*, `range(5)` specifies the range 0 1 2 3 4), the results of commands (*e.g.*, `` `find . -name \*.in -print` ``), the contents of files (*e.g.*, `@input`), and the values of environment variables (*e.g.*, `$DATA_DIR/*.dat`). The last example also shows that words containing a '`*`' or '`?`' will be replaced by files matching the pattern. Sytnax is also available for selecting regular expressions from any of these values.

In addition to the loop variables, several variables (*e.g.*, `%currdir`, `%host`, `%name`) are predefined and will be expanded to their appropriate values upon execution of the loop command. The flag `--macro` allows the you to define other variables appropriate to your expeirments.

When scheduling many experiments at once, one may want to avoid the creation of many individual `labrun` calls, each of which will create its own `.log` file (See Section 3). You can therefore limit the amount of nesting of the loops with the `--nesting` flag. If the value set with this flag is smaller than the number of loops specified, the executable given to `labrun` will itself be a call to `labschedule` containing the remaining loops . This `labschedule` call will not issue calls to `labrun` but will execute the commands give to it directly (achieved through the use of the `--direct` flag). For example,

```
labschedule --for='x y' --for='A B' --nesting=1 bench %1 %2
```

will result in the following two calls to `labrun`

```
labrun --name=schedule-x labschedule -d --nesting=1 --for='A B' bench x %2
labrun --name=schedule-y labschedule -d --nesting=1 --for='A B' bench y %2
```

If the various experiments being scheduled could be run on any one of a cluster of machines, you can specify the names of the machines with the flag `--hosts` and `labschedule` will schedule the tasks on these machines as they become idle. For example, the effect of

```
labschedule --for='10 20 30' --hosts='localhost turing' bench %1
```

is that the following two commands would be issued immediately:

```
labrun --name=schedule-10 ssh localhost cd %curdir; bench 10
labrun --name=schedule-20 ssh turing cd %curdir; bench 20
```

Then, when one of these two runs finshes, the third call to `labrun` for `bench 30` would be issued using `ssh` to the idle machine.

If it is possible to have more than one instance of your experiment running at a time, the flag `--maxtasks` can be used to increase the maximum number of simultaneous experiments per machine. By default, each machine is assigned the number of tasks specified by `--maxtasks` (which is, by default, 1), but it may be desirable to check other conditions (such as the load of the machine) to determine if a host can accept a new task. For this, the flag `--check`, with which you can specify a condition to be checked, is available as well as the variable `%idle` that determines a host's idle percentage and `%check` that determines if a host's idle percentage is above 5.

In the course of running multiple experiments, it may happen that some of them fail for one reason or another. By default, `labschedule` will abort after such a failure. This behavior can be changed (with `--ignore`) such that the remaining experiments will continue to be scheduled. To rerun any failed experiments, it suffices to call `labschedule` once again in the same way it was originally called. The experiments that did not successfully finish will be rerun, but experiments for which a log file exists in which a successful completion is recorded are not rerun. Alternatively, one can indicate that all experiments should be rerun (`--noskip`), and/or that the log files of failed experiments will be preserved (`--keep`).

Further options for this tool allow one to specify the location of the log files (by default, this is `./lab_log`); the prefix of the name to be passed to `labrun` (by default, this is `schedule`); a command other than `ssh %host cd %curdir;` to insert before the `labrun` call; further options to be passed to `labrun`; that the command should be run without using `labrun` or run in the background.

In addition to the log and output files produced by `labrun`, `labschedule` keeps track of its own actions in three files: a `.log` file that logs all relevant actions, a `.out` file that holds the output of all successful runs, and a `.err` file that holds the output of all failed runs. Note that this is in contrast to the meaning of `.out` and `.err` used for `labrun`. The files will be located in the same log directory as the files of `labrun`, and the names will be as follows: `<exp_name>-<date and time>.<ext>`, where `<exp_name>` is `schedule` by default and otherwise the name given as an arugment with the `--name` flag.

3 Running an Experiment

Once the lab environment has been set up, one needs to run the experiment and record the data necessary to be able to reproduce the experiment later. The tool `labrun` accomplishes this. There are three main tasks performed by `labrun`

- it assures that the source code has been "fixed";
- it records the context of the experiment;

- it runs the experiment.

In its simplest form, `labrun` is executed in a directory containing the source code for the experiment's executable. For example:

```
labrun steiner -v closed 41 30
```

will cause the following two things to happen:

- a command `cvs status` is issued in the current directory to assure that the source code in the current directory (which presumably is the source code used to produce the program `steiner`) is up to date.
- if the source is up to date, the program `steiner` will be executed using the arguments `-v closed 41 30`.

The `labrun` tool will generally create at least two files: one that contains the output of the experiment (the `.out` file) and another that records the information about the environment in which the output was created (the `.log` file; see Section 6.3). Additionally, a file with extension `.err` may be generated if the experiment program writes output to the error stream. The `.out` and `.err` files are generated only if the program writes output to the standard output and standard error streams, respectively. The `.log` file is always generated. The names of the generated files all have the same format: `<exp_name>-<date and time>.<ext>`, where `<date and time>` has the format `YYYY-MM-DD-HHMMSS` and `<exp_name>` is, by default, the name of the executable used by `logrun`. By default, these files are created in a subdirectory `lab_log` in the current directory. For the example above, the name of the log file would be something like `./lab_log/steiner-2002-05-21-093412.log`. This file is also accessible as `./lab_log/current.log`, which is a link to the most recent log file. Such a "current" link is created for each of the three possible files generated by `labrun`, thus making it easier to examine the results of the last experiment or to observe the progress of the current experiment as it is running.

Some of the options available for `labrun` allow one to specify

- the location of the CVS directories corresponding to this experiment,
- a different location for the `.log`, `.out`, and `.err` files,
- a name other than the executable to use for the experiment in the created output files,
- a tag other than the `<date and time>` to use in the files created.

For example, altering the above command as follows:

```
labrun --cvs .:/home/hert/src --log ../lab --name example
       --tag closed_41_30 steiner -v closed 41 30
```

will check if the code in the current directory as well as the directory `/home/hert/src` is up to date and, if so, will execute the program `steiner` in the current directory. It will place the created log and output files in a subdirectory `lab` in the current directory's parent directory, creating this subdirectory if it does not already exist. The files created will use `example` for `<exp_name>` and `closed_41_30` instead of

`<date and time>` in the name. Thus the name of the log file for this experiment would be `../lab/example-closed_41_30.log`.

Other options for `labrun` allow one to indicate: that no CVS directories should be checked; that the program should be executed in a different directory; that a specific version of the source code should be used instead of the current one (particularly useful in conjunction with `labmex`); additional things to be recorded in the `.log` file; that all output from the experiment should be recorded in a single file (the `.log` file) instead of spread over possibly three files (the `.log`, `.out`, and `.err` files); and that the experiment should be run in the background and should notify the user upon completion.

Options may also be specified via a local or global resource file (Section 2.1). Such resource files are searched for when processing the input for `labrun` and, if present, the information recorded there will be used to augment the input provided at the command line. In cases of conflict, command-line arguments take precedence over data provided in the local `labrc` file, which takes precedence over data provided in the global `.labrc` file.

4 Rerunning an Experiment

Because one of our goals is to enable experiments to be rerun from the data recorded, we provide a tool, `labrerun`, that does precisely that. In particular, given a log file created by our experimentation tool (or by some other means following the same syntax; see Section 6.3), this tool will run the experiment described in this log file using a context as close to the one described in the original log file as possible. For example, one may rerun an experiment with a command as simple as:

```
labrerun ../lab/example-closed_41_30.log
```

A string of command-line arguments for `labrun` are constructed from the information recorded in the given log file together with the arguments given to `labrerun`. If environment variables were recorded in the input log file, the same values for these variables will be used in the call to `labrun`. If this is not desirable (*e.g.*, because a certain path has changed or software available previously has been upgraded), it is possible, through the `--ignore` option, to tell `labrerun` to ignore the value of an environment variable recorded in the log file and simply record the current value of this variable for the rerun of the experiment. Global and local resource files, if they exist, are ignored by `labrun` when it is called from `labrerun` since all information relevant to an experiment is recorded in its log file and need not be supplemented by data in a resource file.

Once the list of command-line arguments has been constructed and all environment variables are set, the tool will attempt to change to the directory that was current when the log file was created and execute the `labrun` command from there. If that directory no longer exists, the tool will issue a warning and try to proceed in the current directory.

The CVS date tag recorded in the given log file is used to tell `labrun` to check out the version of the code that was current when the original experiment was run. This affects the running of the experiment, of course, only if the experiment involves recompiling the program from its sources. After the rerun of the experiment is finished, the current version of all relevant files under CVS will be restored. If you wish to keep the old files after the rerun, the `--keep` option for `labrerun` can be used. There is also a `--nocvs` option for `labrerun`, which allows you to rerun an experiment with the same settings as an old experiment but using the current version of the source code (*e.g.*, to see the effect of certain code changes).

The name of the log file from which the environmental information was taken is noted in the new log file created for the new run. Environmental differences, if any, can then be discovered using, for example, `diff` with the current log file and the old log file.

The default location of the files created from the experiment is the location of the log file given on the command line, and the files are named in the same way as for `labrun`. A subset of the options available for `labrun` are also available for `labrerun`. They allow one to specify a directory other than the default in which to store the output files, a different directory in which to run the program, a different name to use for the experiment when creating the log file, additional environmental information to be recorded in the new experiment's log file, that all output should be stored in the log file, and that the experiment should be run in the background and notify the user upon completion. There is also an option that allows one simply to print the `labrun` command generated from a log file without executing it and an option to edit the command constructed from the log file before execution.

5 Analyzing the Output

5.1 The *Sus Filter Tools*

Output generated by experimentation programs can come in a myriad of forms. However, the output desired for analyzing or presenting results of experiments generally has one of two forms: a table of values or a graph.

Again, we do not wish to restrict users unnecessarily by insisting on a particular form of output that may be difficult to produce for certain programs. Instead, we provide a set of tools that can process text given in any form to produce a particular internal data format. This internal format, called *sus* (for "Script-readable User Statistics"), is text-based and easily readable by humans (Section 6.4) so users can produce this format directly with their programs if they wish. The tools that convert from the internal format can also process the data by performing mathematical calculations.

Users are asked to specify which data values to extract from a given input file using key words and regular expressions. The *Sus Filter Tools* use the powerful Python language expressions for specifying regular expressions and reformatting data. For details on Python, see `http://www.python.org`.

Two programs, `table2sus` and `text2sus`, convert data to the *sus* format. The first program can be used to convert an ASCII table to the internal format and the second converts a text file in any format to the *sus* format by selecting from the file the data values indicated by the user. For converting from the *sus* format, we provide programs that produce

- an ASCII table (`sus2text`),
- a LaTeX table (`sus2latex`),
- a plot using `gnuplot` (`sus2plot`),

- or another ***sus*** file (sus2sus).

All these programs act as filters by default (reading from standard input and writing to standard output), making it easy to convert any given data file to one of the supported output file formats.

```
cat prog.out | text2sus num_nodes time | sus2plot
```

Alternatively, input and output files can be specified for each program using the `--input` and `--output` options, respectively. For example, the above command could be rewritten as:

```
text2sus --input prog.out num_nodes time | sus2plot
```

Multiple `--input` commands are possible and effect a merging of the input files as if the files were catenated together one after another.

The other command-line options for these tools allow one to specify which data to extract from the file and the format in which to display the output. In the simplest form, the user specifies simply the keyword labels for the desired data, as with the above example. The program will then extract the next word or number after the label as the value to associate with this label in each data record. If you wish to extract a value other than the next word or number to associate with this label, regular expressions can be used on the command line to express the desired value. See the Python documentation or Section 7.7 for more information on forming regular expressions. See Sections 5.2.2 and 5.2.5 for examples.

The tools that convert from a ***sus*** file allow you to manipulate the data in several ways. In particular, one can

1. add a new field (command-line option: `--add`).
   This can be useful to perform mathematical operations on the data values, or to reformat the output for pretty printing.

2. sort the data (command-line option: `--sort`).
   The records are sorted according to the sorting expression. Records with the same key are merged (in accordance with the value given for the `--combine` option). By default, the average of numeric values is taken; string values are always simply concatenated.

3. filter out data (command-line option: `--filter`).
   Only records where the expression gives a nonzero value (not `""` or `0`) are processed.

The switches `--add`, `--sort`, and `--filter` are processed in the same order they appear on the command line. For filtering, sorting, or adding new data fields, the full expresion power of Python can be used. This includes numerical expressions:

```
sus2sus '--filter=float(upperbound) > float(lowerbound)*100'
```

(filter out all records for which the upper bound is more than 100 times the lower bound, the quotes "'" prevent that special characters ("(", ")" and ">" in this example) are interpreted by the shell, see Sections 6.2 and 7.1 for more information on quoting and special characters),
formatting expressions similar to `printf` in C:

```
sus2sus '--add=formattedOpt="%-20f"%float(opt)'
```

(add a formatted `opt` value, left-justified in a field of 20 characters), and string manipulation expressions, using Python's string module:

```
sus2sus '--sort=float(split(time,":")[0])'
```

(sort the records based on the number of hours in a time field with format `HH:MM:SS`. Note the `float`-function that has to be used to sort, e.g. `9` and `10`, in the correct order).

This string module, providing functions such as `join`, `split`, `strip`, and `replace`, is particularly useful for formatting output. We also provide some extensions:

`iff(exp,then,else):`

returns `<then>` if `<exp>` is not empty (`''`, `'0'` or not defined), otherwise `<else>`.

Example:

```
sus2sus '--add=solvedText=iff(solved,"solved","not solved")'
```

`format(val):` Formats a number. Optional parameters:

length: minimal length of the output (negative values produce left-justified output),

digits: number of digits after the decimal point,

mindigits: as digits, but numbers near zero have enough digits to be distinguished from zero,

min: minimal value,

max: maximal value,

pad: padding character, may be:
'0' for zero padding,
'+' for +/-,
'-' for left-justified output.

Examples:
Print `opt` left-justified in a field of minimal length 20 (produces the same result as the example illustrating string formatting above).

```
sus2sus  '--add=formattedOpt=format(opt,length=-20)'
```

Round the time to two digits with a minimum value of 0.01

```
sus2sus '--add=formattedTime=format(time,digits=2,min=0.01)'
```

`savediv(a,b):` Divides `a` by `b`, returning `'division by zero'` if `b` is not a nonzero number or a string representig a nonzero number

Example:

```
sus2sus --add 'gap=savediv(float(upVal)-float(loVal),loVal)'
```

If a variable that is used in an expression is not defined, it will take on '000' as a default value. The switch `--eval` changes this behaviour:

`--eval=strict` undefined variables cause an error,

`--eval=warn` undefined variables produce warnings,

`--eval=debug` print out every expression evaluation, this is very useful for debugging *sus* file transformations

`--eval=invalid` use `'---'` as default (nice for `sus2plot`).

5.2 Examples

The following examples are extracted from the file `sus-tools-demo`, which is distributed with the tool set in the directory `examples`.

5.2.1 `table2sus`

The following table is converted to a *sus* file. The labels of the table are recognized automatically.

Command: `table2sus < steinerTab1.txt > steinerTab1.sus`

steinerTab1.txt

```
instance   size    opt

steiner1  123     123
steiner2  432     434
steiner3  33      44
steiner4  44      33
```

steinerTab1.sus

```
{'labels': ['instance', 'size', 'opt'],
 'table': [{'size': '123', 'instance': 'steiner1', 'opt': '123'},
 {'size': '432', 'instance': 'steiner2', 'opt': '434'},
 {'size': '33', 'instance': 'steiner3', 'opt': '44'},
 {'size': '44', 'instance': 'steiner4', 'opt': '33'}]}
```

A table is found in the input file by looking for the last non-commented, non-blank line in the file and determining how many space-separated columns it contains. Then, if no arguments are given to `table2sus`, it attempts to find the labels for the table by looking for the first line containing the correct number of words that could be labels (i.e. alphanumeric words beginning with a letter). If no labels are found in this way, default labels (col1, col2, etc.) will be used. Leading `#`'s are ignored when looking for the labels.

Thus, the same table would result from this input file and we can use `sus2text` to see the output as a text table.

Command: `table2sus < steinerTab2.txt | sus2text` steinerTab2.txt

```
# gnuplot file of data
#
# generated 12-Jul-2002 14:32:04
#
#instance   size    opt

steiner1  123     123
steiner2  432     434
steiner3  33      44
steiner4  44      33
#
# ignore me
```

output

```
instance size opt

steiner1 123  123
steiner2 432  434
steiner3 33   44
steiner4 44   33
```

5.2.2 text2sus

Given a simple text file, a table can be constructed by giving simply labels or labels followed by regular expressions. The following input file is scanned for `instance`, `running_time`, `value`, and something more complex: the last word of a line with `more` at the beginning. This word gets the label `complex`. Additionally, we want the values of `time` as `hours` and `minutes`.

Note that the order of the labels does not matter and it is not necessary to have all lables in each of the records. An exception is the first label: It defines when to start a new record. (Alternatively one could use the `--next` switch.)

Command:

```
text2sus instance running_time value 'complex=^more.* (\w+)'
    'time: (?P<hours>.*):(?P<minutes>.*)' < text2sus.txt >
    text2sus.sus
```

The reason for the ' ' in this command is that otherwise the shell gets confused by the * and the spaces.

text2sus.txt

```
instance: steiner1.stp
time: 11:23
blablabla
more complicated, we want the last word in this line! 23
value: 123
```

```
running_time: 123

instance: steiner2.stp
more 44
time: 11:44
value: 312
running_time: 323

instance: steiner3.stp
time: 11:55
more 45
value: 32
running_time: 532

instance: steiner4.stp
time: 12:04
more 46
value: 44
running_time: 954
```

text2sus.sus

```
{'labels': ['instance', 'running_time', 'value', 'complex', 'hours',
   'minutes'],
 'table': [{'running_time': '123', 'value': '123', 'minutes': '23',
   'complex': '23', 'instance': 'steiner1', 'hours': '11'},
 {'running_time': '323', 'value': '312', 'minutes': '44', 'complex':
   '44', 'instance': 'steiner2', 'hours': '11'},
 {'running_time': '532', 'value': '32', 'minutes': '55', 'complex':
   '45', 'instance': 'steiner3', 'hours': '11'},
 {'running_time': '954', 'value': '44', 'minutes': '04 ', 'complex':
   '46', 'instance': 'steiner4', 'hours': '12'}]}
```

5.2.3 merging files and adding fields

This example features merging of different *sus* files and creating new data fields (columns) in the table.

Command:

```
sus2text --sort instance --add solved=opt==value --input
    steinerTab1.sus --input text2sus.sus instance running_time
```

The input values come from `steinerTab2.sus` and `text2sus.sus`, the output of the two previous examples. As index variable, the label `instance` (common to the two input files) is used. The additional label `solved` is introduced, set to the value `opt==value`, where `opt` is a label in `demofile1.sus` and `value` a label in `demofile2.sus`.

output

```
instance running_time solved

steiner1 123          1
steiner2 323          0
steiner3 532          0
steiner4 954          0
```

5.2.4 filtering

When converting from a *sus* file to another format, certain records can be filtered out using the `--filter` option.

Command: `sus2text '--filter=instance<"steiner3"' < text2sus.sus`
The ' ' are needed because of the `"` and `<`, which would be interpreted by the shell otherwise.

```
output
instance running_time value complex hours minutes

steiner1 123          123   23      11    23
steiner2 323          312   44      11    44
```

For comparison, the unfiltered output is:

Command: `sus2text < text2sus.sus`

```
output
instance running_time value complex hours minutes

steiner1 123          123   23      11    23
steiner2 323          312   44      11    44
steiner3 532          32    45      11    55
steiner4 954          44    46      12    04
```

5.2.5 command option file

This example features an option file and different ways to define new labels. Notice that the table in the output is not a complete table. The last two entries don't really have a `lastword` because the last word is used as the value for the `free:` label already. The default value of 000 is used for the third label in these entries.

Command: `text2sus @testfree.dat < testfree.txt | sus2text`

```
testfree.data
# This is a command-line file for text2sus
# Each option is defined in a line, empty lines or lines
# beginning with "#" are ignored.
```

```
--next=\n
# This indicates that each line is a separate record

test
# We want to parse for the keyword test.

freetext=free:\s*(\w+)
# And for the first word after "free:", the label for this will be
# "freetext".

(?P<lastword>\w+)$
# And for the last word in the line, getting the label "lastword"
```

testfree.txt

```
test: 123, free: sadds, ewrwre
test: 444, free: dfgf, trwret
test: 123, free: sadds
test: 444, free: dfgf
```

output

```
test freetext lastword

123  sadds    ewrwre
444  dfgf     trwret
123  sadds    000
444  dfgf     000
```

5.2.6 sorting and combining values in different ways

When converting from a *sus* file to another format, by default all entries are extracted in the order given. If one wishes to sort the entries according to the value of a certain expression, multiple entries with the same value must be combined somehow. By default, the average of the numeric values is taken (string values are simply concatenated).

Command: `table2sus < graph.out | sus2text --sort ’float(vertices)’`

graph.out

```
#vertices edges run1  run2

10 20 123.6   141.3
20 80 2321.4  842.9
10 40 432.8   832.0
20 40 943.1   314.2
```

output

```
vertices edges run1    run2

10.0     30.0  278.2   486.65
20.0     60.0  1632.25 578.55
```

One can combine multiple entries using the minimum, maximum, mean, sum, or product of the values using the `--combine` option. Here we take the minimum.

Command:

```
table2sus < graph.out | sus2text --sort 'float(edges)'
    --combine min
```

output

```
vertices edges run1   run2

10       20    123.6  141.3
10.0     40.0  432.8  314.2
20       80    2321.4 842.9
```

One can also sort the entries according to some expression. Here we sort the records based on the total time for both runs.

Command:

```
table2sus < graph.out | sus2text
    '--sort=float(run1)+float(run2)'
```

output

```
vertices edges run1   run2

10       20    123.6  141.3
20       40    943.1  314.2
10       40    432.8  832.0
20       80    2321.4 842.9
```

5.2.7 adding fields with formatting

This example shows some of the possibilities for reformatting data. An additional field `avg_time`, which is a average of two existing fields `run1` and `run2`, is added to each record. Since all values in a ***sus*** file are stored as strings, a conversion to a number is necessary before using these in arithmetic expressions. The computed value is formatted in a field of width 6 with only one digit after the decimal point. This is done using the built-in string formatting from Python, which is similar to `printf` in C.

Command:

```
table2sus < graph.out | sus2sus '--add=avg_time="%6.1f" %
    ((float(run1)+float(run2))/2)' > avgrun.sus
```

avgrun.sus

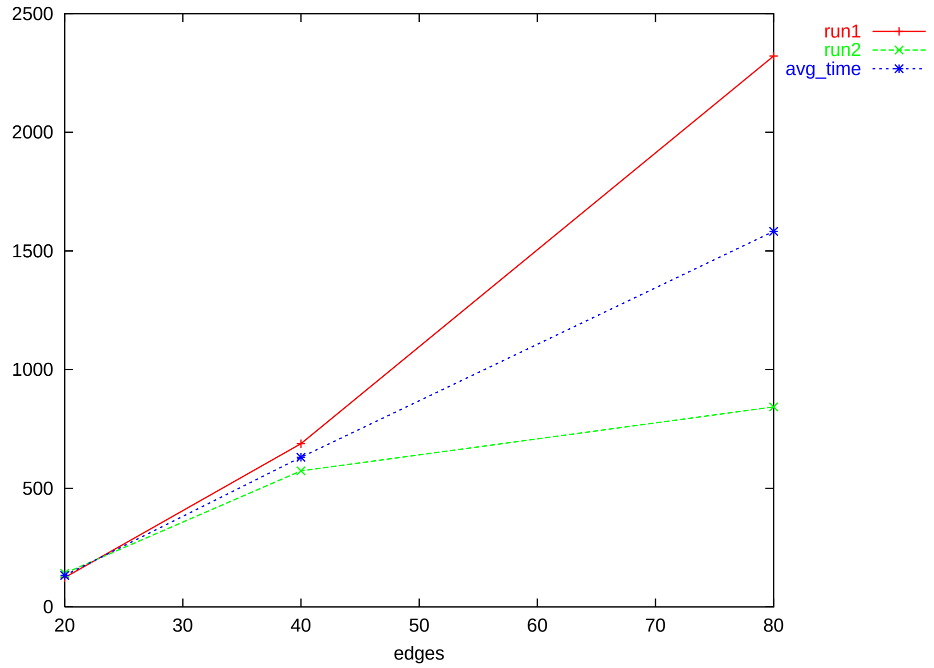
```
{'labels': ['vertices', 'edges', 'run1', 'run2', 'avg_time'],
 'table': [{'vertices': '10', 'avg_time': ' 132.4', 'run2': '141.3',
   'edges': '20', 'run1': '123.6'},
 {'vertices': '20', 'avg_time': '1582.2', 'run2': '842.9', 'edges':
   '80', 'run1': '2321.4'},
 {'vertices': '10', 'avg_time': ' 632.4', 'run2': '832.0', 'edges':
   '40', 'run1': '432.8'},
 {'vertices': '20', 'avg_time': ' 628.6', 'run2': '314.2', 'edges':
   '40', 'run1': '943.1'}]}
```


5.2.8 `sus2plot`

Now we plot the values in the input file used in the previous example. We plot `run1`, `run2`, and `avg_time` against `edges`. The values of several lines with the same number of edges are combined by taking the average. The option `--combine` can be used to provide another combining rule as illustrated previously.

Command:

```
sus2plot --sort 'float(edges)' run1 run2 avg_time <
    avgrun.sus
```



5.2.9 `sus2latex`

This simple example illustrates what the tables produced by `sus2latex` look like.

Command: `sus2latex < avgrun.sus`

output

```
%%sus2latex, Revision 1.105, 2002/11/14
% \begin{tabular}{|l|l|l|l|l|}
% vertices & edges & run1   & run2  & avg_time \\ \hline
```

```
10 & 20 & 123.6  & 141.3 &  132.4 \\
20 & 80 & 2321.4 & 842.9 & 1582.2 \\
10 & 40 & 432.8  & 832.0 &  632.4 \\
20 & 40 & 943.1  & 314.2 &  628.6 \\

% \end{tabular}
```

6 File Formats

6.1 `labrc` Files

The format of the two configuration files (`~/.labrc` in the user's home directory, and `labrc` in the current working directory) created by `labsetup` and read by the tools is as follows:

- Lines beginning with `#` and empty lines are ignored.
- Lines beginning with `+` are joined with the previous line.
- Different options are placed on different lines.
- The files can have different sections for the different programs. The sections are separated by lines containing the name of the program in square brackets (*e.g.*, `[labrun]`). The first section, before the first separator, contains global settings for all programs.

For each tool, it is possible to tell the tool to ignore the `~/.labrc` and `./labrc` file by using the switch `--nolabrc`.
Example:

`~/.labrc`

```
# global settings for all tools
--verbose

# end of global settings

[labrun]
--comment=A comment from ~/.labrc

[labmex]
--Make=gmake
```

Switches and arguments for the `lab`-commands are searched for in three places:

1. the command line,

2. a `labrc` file in the current directory,
3. a `.labrc` in the user's home directory.

Usually, switches found first override switches found later, *e.g.*, a `-x ~/foo` in `~/.labrc` is overridden by a `--exec=~/bar` from the command line. Exceptions are switches that accumulate (e.g., the `--comment`, `--cvs`, and `--env` switches of `labrun`). They are just collected from all sources.

6.2 Command Input Files

As some of the tools provide a lot of switches and parameters, it is possible to store the command-line arguments in files and load them into the command line with `@<filename>`. This also has the advantage that there is no need to quote and escape special characters that are usually interpreted by the shell.

A command input file has the same format as the configuration files (Section 6.1). Additional notes:

- The input read from the file simply replace `@<filename>` in the command line. Thus, it is possible to specify both switches and arguments in command input files.
- Command input files may be nested.
- Every line yields a single argument of the command line.
- Each argument is passed to the tools as a single "word" with spaces in the argument escaped to glue the word together and special characters escaped to preserve their integrity. Two exceptions are:
  1. Switches that require a value:
     (See 7.1 for more information about the format of command-line switches.)
     Usually one must put the value directly after the short switch name:

     ```
     -lnew_log
     ```

     or after an `=` when using the long switch name:

     ```
     --log=new_log
     ```

     For convenience, white space is allowed in command input files between the name of the switch and its value:

     ```
     -l new_log
     --name new_name
     ```

  2. The first line that is not a switch:
     If this line contains spaces, it will be split into several words (see example below).

Example:

labrun.opt

```
# command-line options for labrun
--cvs=~/steiner_c
--log=~/tmp/exptest
--exec=~/steiner_c
--name=test

--env=LD_LIBRARY_PATH
--comment=tagtest2
--comment=EnvTest=$PATH
--comment=FileTest=@~/.cshrc:PATH\s+(.*)
--comment=ExecTest=`head -1 ~/.cshrc`:.*\s(.+)
-v
steiner
 grid_solve.str
 problem=dmxa1200.stp
# There are different ways to specify the command to be executed. You
# need not indent it. And you can put everything on one line:
#
#       steiner grid_solve.str problem=dmxa1200.stp
#
# As it is the first argument for labrun and it contains spaces, this
# line will be split into three lines. The following is equivalent:
#
#   steiner grid_solve.str
#   + problem=dmxa1200.stp
#
# Note that only the first line is split. Thus, the following won't
# work; it calls "steiner" with one argument containing a space.
#
#   steiner
#   grid_solve.str problem=dmxa1200.stp
#
# Because we don't want to have the same mess with special characters
# as with the shell, the following is illegal in command-line files
# and will be rejected:
#
#   steiner 'some argument containing a space' 'another argument'

NEX
# with NEX starts the next experiment
-v
--cvs=~/steiner_c
--log=~/tmp/exptest
--exec=~/steiner_c
```

```
--name=grid_solve2
steiner grid_solve2.str problem=dmxa1200.stp
```

Notice that, though section separators (e.g., [labrun]) are possible in command input files, they are not necessary if the file contains options for only a single tool since all commands before the first separator are used by all tools.

6.3 .log Files

6.3.1 labrun

The .log file generated by labrun is a plain text file. It should be possible for anyone reading the file to understand it immediately, but a particular format is required by the labrerun tool, and this is described here.

A .log file consists mainly of a set of label-value pairs. Labels and values are both character strings and a colon is used to separate a label from its value. This implies that labels cannot contain colons in their names; values may however. Values can be spread over multiple lines by using a + at the beginning of continuation lines.

A value may be followed by an indication of how the value was produced (to facilitate reproducing this value from the labrerun tool). This means, the following three formats of label-value pairs are possible in a .log file:

1. label: value
2. label: value {cmd}
3. $label: value

Formats 2 and 3 are for labels provided exclusively by the user (either on the command line or via a configuration file). Format 3 is used for environment variables recorded in response to the --env option for labrun. Format 2 is for all other kinds of comments. Pairs using format 1 may be provided by the user or by the labrun tool. The following table lists the labels reserved for use by labrun together with explanations of their values.

label	explanation
`labrun Release`	release number of labrun
`Start date`	format: weekday month day HH:MM:SS timezone year
`OS`	operating system
`Hardware`	hardware information
`Machine`	hostname
`Processor`	speed and name
`Memory size`	base memory size
`L2 data cache`	L2 data cache size
`CVS dirs`	colon-separated list of directories checked with CVS
`CVS date tag`	current date stamp
`Previous logs`	present if the run was produced by `labrerun`
`Command`	command line
`Exec dir`	execution directory
`Current dir`	working directory
`Output file`	name of the output file
`Error file`	name of the error file
`Error level`	exit code if command failed
`User time`	estimate of the user time spent for the command (in seconds)
`Stop date`	format: weekday month day HH:MM:SS timezone year

Comments, which will be ignored by `labrerun`, can be added to a log file in the following ways.

- All lines beginning with a `#` are regarded as comments and ignored.
- All lines following a line containing only hyphens are ignored. (This is useful for adding long comments at the end of a file or for using `labrerun` with a log file created using the `--one` option for `labrun`.)

Example:

sample.log

```
labrun Revision: 1.92

Start date:        Fri Aug  2 15:49:42 CEST 2002
OS:                Linux 2.2.19 #4 Tue Aug 7 16:06:42 CEST 2001
Hardware:          i686
Machine:           mpino1109
Processor:         225 MHz Pentium III (Coppermine)
Memory size:       257920 kB
L2 data cache:     256 KB

Compiler version: 3.1                   {‘$CC -v‘:version}
$CC:               gcc
$CXXFLAGS:

CVS dirs:          .
```

```
CVS date tag:      2002-08-02 13:49 GMT

Command:           labmex '--Make=gmake' --clean=both --log=lab_clog
+                  --Make=gmake sort-demo 1000 10
Exec dir:          /home/hert/talks/tool_set/tutorial/sort
Current dir:       /home/hert/talks/tool_set/tutorial/sort
Output file:       ./lab_log/sort-demo-2002-08-02-154942.out

User time:         0.9 sec
Stop date:         Fri Aug  2 15:49:44 CEST 2002
```

6.3.2 `labschedule`

The log file produced by `labschedule` has a format similar to that produced by `labrun`. It contains, however, information relevant only to the scheduling of the different tasks. In uses therefore a different set of labels and values as described in the following table.

label	explanation
`labschedule Release`	release number of `labschedule`
`Start date`	format: weekday month day HH:MM:SS timezone year
`Hosts`	the set of hosts `labschedule` had to choose from
`Current dir`	the directory in which `labschedule` was called
`Output file`	the output file, containing output of successful experiments
`Error file`	the error file, containing output for unsuccessful experiments
`Command line`	the `labschedule` command
`Command`	the executable command for the tasks scheduled
`Stop date`	format: weekday month day HH:MM:SS timezone year

Between `Command` and `Stop date` there are two lines for each experiment that is run with the following format:

```
HH:MM:SS:    <host>: started '%1-%2-...'.
HH:MM:SS:    <host>: finished '%1-%2-...' [<error message>].
```

where `HH:MM:SS` is the time a particular job was started or finished and the variables %1, %2, *etc.* are substituted with their values for each particular experiment. If an experiment fails, a message containing its error code and the command that caused this failure will be repeated. Experiments that are skipped are also so noted in the log file.
Example:

`schedule-sample.log`

```
labschedule Release: 0.6

Start date:     Thu Nov  7 14:59:23 CET 2002
Hosts:          mpino1109
Current dir:    ~/tool_set/explab/example/tutorial
```

```
Output file:   ./lab_log/schedule-2002-11-07-145923.out
Error file:    ./lab_log/schedule-2002-11-07-145923.err
Command line:  /home/hert/bin/labschedule -f range\(100,200,50\) -f
+              10\ 20 labmex sort-demo %1 %2

Command:       labrun --log=/home/hert/example/tutorial/lab_log
+              --name=schedule-%1-%2  labmex sort-demo %1 %2

14:59:23:      mpino1109: started '100-10'.
14:59:28:      mpino1109: finished '100-10'.
14:59:28:      mpino1109: started '100-20'.
14:59:32:      mpino1109: finished '100-20'.
14:59:33:      mpino1109: started '150-10'.
14:59:38:      mpino1109: finished '150-10'.
14:59:38:      mpino1109: started '150-20'.
14:59:42:      mpino1109: finished '150-20'.

Stop date:     Thu Nov  7 14:59:43 CET 2002
```

6.4 *sus* Files

The *sus* (standing for "Script-readable User Statistics") file is a text file with a Python-syntax dictionary with two entrys:

table This is a list of dictionaries. Each dictionary represents one data record, which consists of *label*:*value* pairs. All labels and values are strings (enclosed in single quotes).

labels This is a list of labels used in the table. It is used mainly to specify the order of the labels, as the keys of a dictionary are not ordered.

There may be some 'newline' characters in the *sus* file to improve readablility.

When writing a *sus* filewith `sus2sus`, `text2sus` or `table2sus`, one can add the switch `--binary`. The *sus* filewill then be written in a binary format. The format is produced by the python standard module "pickle" and is much faster to parse by the computer, but unreadable for humans.

Example:

test.sus

```
{'labels': ['instance', 'size'],
 'table': [{'size': '450', 'instance': 'test1'},
 {'size': '694', 'instance': 'test2'},
 {'size': '90', 'instance': 'test3'}]}
```

is a *sus* file representing the table

instance	size
test1	450
test2	694
test3	90

7 The Current Tools

7.1 Command-line Options

Each of the tools in our set has a command-line interface following the style of other common Unix tools and also allows command-line options to be specified in a file (Section 6.2), whose name is given on the command line. As for most Unix tools, almost all command options have both long and short forms. The short forms start with a single “`-`”, the long ones with two. Some options take additional parameters. With the short form, these parameters can be put directly after the command option (“`-nnewname`”) or can be separated from it by a blank (“`-n newname`”). With the long form, there may be a “`=`” (“`--name=newname`”) or a blank (“`--name newname`”) separating the parameter from the command option. The long forms may be abbreviated (“`--na=newname`”) as long as the prefix is unique among the switches available for the tool in question.

If some parameters contain special characters, such as “`*`”, “`(`” or “`"`”, this can lead to conflicts with the shell, which tries to interpret these characters in its own way. The interpretation of a character can be prevented in most shells, by either putting a “\” before the character or by quoting the word (or parts containing the special characters) with “’”. For example:

```
sus2sus '--filter=int(points)>100' --add quad=simp\*simp
```

Notice that when an option is enclosed in quotes, the “`=`” separator between the option name and the value is necessary.

Another way of getting around the problem of these special characters is the use of command input files (Section 6.2). Every character is such a file is treated literally, which means that special characters need not (indeed must not) be quoted. You can insert the parameters of a command inpu file at any point in the command line using “`@ <filename>`” or “`@<filename>`”. All styles of input on the command line can be mixed:

```
labrun --cvs=~/dir --name test -ccomment1 -c comment2 @optfile
```

All `lab`-programs start other programs. This gives rise to the additional problem that when passing the commands via a shell command, the special characters have to be quoted again. For this reason the characters “`[]<>()$␣*?\"'\|;&`” are quoted by a backslash once read in by the tools. If you want the characters to be interpreted by a shell, you have to include a “`sh -c`” command explicitly. (See page 40 for an example.)

7.2 `labsetup`

Purpose:

Generate an ASCII configuration file that stores information about the context of a certain (set of) experiment(s).

Usage:

`labsetup [<options>]`

Options:

`-h, --help` print a help message

`-l, --local` create the file `labrc` in the current directory instead of `.labrc` in the user's home directory.

`-v, --verbose` verbose mode

`--version` print version information

Description:

When the program is run, the user will be asked to provide information about the lab environment that is being set up. By default, this is the global environment, and for each question asked the current value represented in any existing `($HOME)/.labrc` file will be displayed. If no value is set in this file, the default value will be displayed. The user can then accept the current (possibly default) value or change the value as desired. Changing back to a default value can be done using the string "use default" as the answer to a non-yes/no question.

When the `-l` option is used, indicating that a local environment is being set up, the values displayed with each question represent the settings from the global resource file as well as any `labrc` file in the current directory with the settings in the local file taking precedence over those in the global file when conflicts arise (*i.e.*, the settings `labrun` would use). Questions corresponding to options that are set in the global file and cannot be changed in the local file will not be asked.

Result:

A file containing the corresponding command-line options for the tools. By default, the file is called `($HOME)/.labrc`.

7.3 labrun

Purpose:

Tool for running benchmarking or other non-interactive tests

Usage:

`labrun [<option>] ... <program> [<argument>] ...`

Options:

`-a, --autocvs` automatically `'commit'` and `'update'` CVS if necessary

`-b, --batch` run in background

`-C, --cvs=<DIRS>` check CVS in `<DIRS>` (colon-separated lists and multiple entries are allowed) (default: `'.'`)

`--nocvs` do not check CVS

`-c, --comment=<TAG>` specifies a tag that will be recorded in the .log file. The format of `<TAG>` is:

`[ <LABEL> = ] <STRING>`

Inside the `<STRING>` the following expressions are possible:

`$<ENVVAR>` will be replaced by shell variable `<ENVVAR>`,

`@<FILE>` will be replaced by file `<FILE>`,

`` `<COMMAND>` `` will be replaced by output of `<COMMAND>`.

`%` will be replaced by the output file's name

Each of the first three statements may be followed by a modifier:

`:<PATTERN>`

If the regular expression `<PATTERN>` contains parentheses, only the match corresponding to the parentheses will be taken. Otherwise the first word after the regular expression will be taken. The `<PATTERN>` may not contain spaces. Use `\s` instead. Some documentation for regular expressions is given in Section 7.7. If `<STRING>` begins with `0x`, the number following this will be converted from hexidecimal to decimal.

Examples:

```
OS=`uname -srv`
Processor Speed=@/proc/cpuinfo:cpu.MHz MHz
Processor Name=It's a @/proc/cpuinfo:model.name[\s:]+(.*)
Solution=@%:final.value
```

`-d, --date=<DATE>` use the CVS versions current on the given date (`<DATE>` should be in a format understandable by CVS)

`--keep` keep old files (in conjunction with `--date`)

`-e, --env=<ENV>` record the value of environment variable `<ENV>` (will be restored when using `labrerun`.)

`-h, --help` print a help message

`--help-comment` print help message for formulating comments

`-i, --info` just give environment info (and evaluate `--comment` switches)

`-l, --log=<DIR>` where to put log files (default: `'./lab_log/'`)

`-n, --name=<TAG>` name to use instead of program name in file names

`-1, --one` just produce one `.log` file with different sections

`-t, --tag=<TAG>` use `<TAG>` instead date-time stamp in file names

`-v, --verbose` verbose (multiple `-v` increases verbosity)

`--version` print version information

`-x, --exec=<DIR>` where to execute program (default: `'.'`)

`--nolabrc` do not read `~/.labrc` and `./labrc`

With `@FILE` or `@ FILE` (some) command-line options are read from `FILE` (see section 6.2).

As mentioned in Section 2.2.2, the keyword `NEX` (for "new experiment") can be used in the command line to indicate that what follows is the input for a new experiment. For an example of this, see Section 6.2.

Result:

Files `<NAME>-<TAG>.log` and `<NAME>-<TAG>.out` will be produced in the log directory (unless the `--one` option is used). If there was some output to standard error, a file `<NAME>-<TAG>.err` will also result. For each of the `.out`, `.log` and `.err` files created, a link will be created in the log directory by the name of `current.<ext>`.

Examples:

- `labrun steiner -v closed 41 30`

  Executes the program `steiner` with arguments `-v closed 41 30`, creating a log file and an output file in the default log directory `lab_log`.

- ```
  labrun --cvs .:/home/hert/src --log ../lab --name example
          --tag closed_41_30 steiner -v closed 41 30
  ```

  (The command should be all on one line, of course.) Overrides the default CVS directory (adding a directory in addition to the current one), the default log directory (using `../lab` instead of `./lab_log`) and the default file name prefix (using `example-closed_41_30` instead of `steiner-<date-time>`).

- `labrun -e CC -c 'Compiler version=`$CC -v`:version\s+(\S*).*' timings`

  Records the value of the environment variable `CC` (the compiler) and the version number of this compiler in the log file. Notice that the version number is achieved by executing the command `$CC -v` and then looking for the first set of non-spaces following the word "version" in the resulting text. Notice also the different quoting required. The quotes (') around the argument to `-c` are required because of the space in the label for the value and the stars used in the regular expression. The quotes (`) around the command are required to indicate that it is a command to be executed.
  ```

- `labrun -n timings labmex -x benchmark -m CC=gcc ~/src/timings`

  Uses `labrun` in conjunction with `labmex`. Without the `-n` argument to `labrun`, the name of the experiment would be `labrun`, which is probably not what you want. See Section 7.5 for explanation of the `labmex` arguments.

- `labrun ssh turing labmex fun 123 NEX ssh oracle labmex fun 123`

  Runs two experiments. The first compiles the program `fun` on the machine `turing` and then executes it with arguments `123`. The second experiment compiles the same program on the machine `oracle` and executes it using the same arguments. Without additional arguments all commands are executed in the user's home directory on these two machines.

- `labrun ssh turing 'cd benchmark; labmex fun 123'`

  Runs the experiment above on the machine `turing` in the directory `benchmark`.

7.4 labrerun

Purpose:

rerun an experiment using information recorded in a `.log` file

Usage:

`labrerun [<option> ...] <log file>`

Options:

`-b, --batch` run in background

`-c, --comment=<TAG>` will be stored in the .log files (see '`labrun --help-comment`' for details)

`-e, --env=<ENV>` record the value of environment variable `<ENV>` (will be restored, when using `labrerun`)

`-h, --help` print a help message

`-i, --ignore=<ENV>` ignore the value of environment variable `<ENV>` recorded in the log file; use the current setting instead

`-l, --log=<DIR>` where to put log files (default: location of `<log file>`)

`-n, --name=<TAG>` name to use instead of program name in file names

`-t, --tag=<TAG>` use `<TAG>` instead date-time stamp in file names

`--keep` keep old files that were checked out to rerun the experiment

`--nocvs` do not revert to CVS version recorded in log file; use current version instead

`--nolabrc` do not read `~/.labrc` and `./labrc`

`-1, --one` just produce one .log file with different sections

`-p, --print` print the labrun command that would be created but don't execute it

`-v, --verbose` verbose mode

`--version` print version information

`-x, --exec=<DIR>` where to execute program (default comes from log file)

With `@FILE` or `@ FILE` (some) command-line options are read from `FILE` (see section 6.2).

Result:

The same as for `labrun`.

Examples:

- `labrerun lab_log/current.log`

  Reruns the last experiment

- `labrerun --nocvs lab_log/current.log`

  Reruns the last experiment using the current version of the code

- `labrerun -e LD_LIBRARY_PATH -l ../log -n new -t 20_5 lab_log/current.log`

  Reruns the last experiment (`lab_log/current.log`) recording the value of the environment variable `LD_LIBRARY_PATH` in the new log file. The log file is stored in the directory `../log` and has the name `new-20_5.log`.

- `labrerun -i LD_LIBRARY_PATH ../log/new-20_5_10.log`

  Reruns the previous experiment, using the current value of `LD_LIBRARY_PATH` instead of the one recorded in the log file.

7.5 labmex

Purpose:

make and execute a program

Usage:

`labmex [<option>] ... <executable> [<arguments>]`

Options:

`-c, --clean=[before|after|both]` do a '`make <make_options> clean`' before, after, or before and after compiling the target

`-h, --help` print a help message

`-l, --log=<DIR>` specifies the directory in which the compile log will be created (default is `'.'` or, if called by `labrun`, the directory in which labrun's log files are stored). This log file is not created if compilation succeeds.

`-k, --keep` always keep the compile log

`-m, --makeflag=<OPTION>` pass the given option to each call of make; use multiple `-m` options for multiple make options

`-M, --Make=<COMMAND>` use `<COMMAND>` instead of "make" when compiling the executable

`--nolabrc` do not read `~/.labrc` and `./labrc`

`-t, --target=<TARGET>` target for make (default is `<exectuable>`); multiple `-t` options are allowed

`-v, --verbose` verbose mode

`--version` print version information

`-x, --exec=<DIR>` where to execute `<executable>` (default is `'.'`)

With `@FILE` or `@ FILE` (some) command-line options are read from `FILE` (see section 6.2).

Result:

Program is compiled and run. If the `--keep` option is used or the compilation fails, a compile log and a link to this file called `current.clog` is created in the current directory, the log directory of `labrun` if called from `labrun`, or the log directory specified on the command line.

Examples:

- `labmex --clean both timings 10 20 5`

  Performs a `make clean` and then a `make timings`. Runs the program `timings` with arguments `10 20 5`. Then does a second `make clean`.

- `labmex -x benchmark -m -DCXXFLAGS=-O3 -t timings ~/src/timings`

  Performs a `make -DCXXFLAGS=-O3 timings` in the current directory. Moves to the directory `benchmark` and executes the command `~/src/timings` in that directory.

- ```
  labmex -M gmake -m -C.. -m '-DCXXFLAGS=-NDEBUG -NDEBUG2'
       -x .. -t all timings
  ```

  Performs a `gmake -C.. '-DCXXFLAGS=-NDEBUG -NDEBUG2' all` command in the current directory, which compiles the target `all` in the parent directory (the `-C..` option of `gmake`). Moves to the parent directory and executes the command `timings`.
  ```

7.6 `labschedule`

Purpose:

starts multiple commands (possibly on multiple hosts or on a multiprocessor environment)

Usage:

`labschedule [<option>] ... <executable> [<arguments>]`

Options:

`-f, --for=<WORDLIST>` loop through all words from `<WORDLIST>`; in the command line the corresponding loop variable (%1, %2, ...) will be replaced by words from `<WORDLIST>`. (See the 'for-loop' section below for details.)

`--nesting=<N>` only the first `<N>` loops will form individual `'labrun'` calls; the remaining loops will incorporated into one experiment (with executable `labschedule --direct ... <executable> ...`)

`--macro=<LABEL=WORDLIST>` define macro (`%LABEL` will be expanded to `WORDLIST`)

`--hosts=<HOSTLIST>` execute on hosts `<HOSTLIST>` (default: localhost)

`--maxtasks=<N>` maximum tasks started on one host (default: 1)

`--check=<EXP>` condition to test to determine if host can accept another task (typically for multi-user hosts; see the 'check' section below)

`-n, --name=<TAG>` `<TAG>` for log files (default: `'schedule'`)

`-l, --log=<DIR>` `<DIR>` for log files (default: `./lab_log`)

`-i, --ignore` ignore error codes from failed experiments, continuing with the rest of the loop (default: abort)

`--keep` keep old (incomplete, failed) log files (default: delete them)

`--noskip` don't skip previously performed runs (default: skip)

`--prefix=COMMAND` prefix for each command (default: `ssh %host; cd %curdir`)

`--nolabrun` don't use 'labrun `--log=%logdir --name=%name-%1-%2-...`' as command executed

`--labrunflag=<OPTION>` additional labrun option; use multiple `--labrunflag` options for multiple labrun options

`-d, --direct` execute loops directly (see 'direct' section below)

`-b, --batch` run in background

`-p, --print` do not perform the commands, just print them

`-v, --verbose` verbose mode

`--nolabrc` do not read `~/.labrc` and `./labrc`

With `@FILE` or `@ FILE` (some) command-line options are read from `FILE` (see section 6.2).

Result:

For every combination of the words in each of the `--for` commands, an experiment is started. In addition to the log and output files produced by `labrun`, `labschedule` keeps track of its own actions in three files: a `.log` file that logs all relevant actions, a `.out` file that holds the output of all successful runs, and a `.err` file that holds the output of all failed runs.

Defining For-Loops: The word lists for the for-loops can be defined in several ways. The following expressions are possible:

`$<ENVVAR>` will be replaced by shell variable `<ENVVAR>`,

`@<FILE>` will be replaced by file `<FILE>`,

`` `<COMMAND>` `` will be replaced by output of `<COMMAND>`.

Each of the three statements may be followed by a modifier:
`:<PATTERN>`
If the regular expression `<PATTERN>` contains parentheses, only the match corresponding to the parentheses will be taken. Otherwise the first word after the regular expression will be taken. The `<PATTERN>` may not contain spaces. Use `\s` instead. Some documentation for regular expressions is given in Section 7.7.

Furthermore, the characters `~`,`*` or `?` in words will be interpreted as a shell would.

`eval(EXPRESSION)` will be replaced by the evaluation of the python `EXPRESSION`, `range(X)` is a shortcut for `eval(range(X))` and will be replaced by the words $0, 1, 2, 3, \ldots, (\mathtt{X} - 1)$.

Expansion of Variables: Some variables are expanded before executing a command. These can be used in `--check`, `--labrunflag`, `--prefix`, `--for`, and in the arguments to the experiment.

`%curdir` the current directory

`%logdir` full path to the log directory

`%host` the host for the current experiment

`%prefix` either `ssh %host cd %curdir;` or an empty string

`%name` the name, provided by `--name` (default: `schedule`)

`%maxtasks` number of tasks per host, provided by `--maxtasks`

`%nesting` nesting level, provided by `--nesting`

`%1, %2, ...` will be replaced by the current word from the first, second, and so on `--for` word list. Here one can use modifiers similar to those of tcsh, (e.g., `%1:h`). Example (if `%1` gives `/dir/name.ext`):

h)ead (directory)	%1:h	/dir
t)ail (name)	%1:t	name.ext
e)xt (extension)	%1:e	ext
r)oot (skip ext)	%1:r	/dir/name
s)ubst (regular expression substitution /from/to/, any character may be delimiter)	%1:s-e(.)-\1E-	/dir/nam.ExEt

More variables can be defined by the --macro=LABEL=WORDLIST option. Valid LABELs are alphanumeric and begin with a letter. Two predefined macros are:

- %idle=‘%prefix vmstat 1 2‘:(?s).*\D(\d+)
  reads the host’s idle percentage (useful for --check) ((?s) makes .* match also newlines, \D is [^0-9] and (\d+) returns the last number printed by vmstat.)
- %check=%idle > 5
  (useful for --check=%check on multi-user systems, see below)

Checking Idle Time:

By default, every host gets assigned %maxtasks tasks. On multiuser systems it may be advisable to check if the host can accept a further task. This can be done with the --check switch:

--check=<EXP> expression to check, if host can accept another task.

The results of commands, contents of files, and values of environment variables can be incorporated into EXP in the same way they can for for loop values. (See two sections above).

Example:

- --check=‘ssh %host w‘:load.average < %maxtasks
  reads the load of the host using the w command.
- --check=‘ssh %host vmstat 1 2‘:[^@]*([0-9]+) > 5
  --check=‘%prefix vmstat 1 2‘:[^@]*([0-9]+) > 5
  --check=%idle > 5
  --check=%check
  reads the idle percentage of the host using the vmstat command (all variants resolve to the same command).

Using the --direct switch: The --direct switch is used internally for performing multiple runs of a program as one experiment when --nesting is used. If both --direct, and --nesting=<LEVEL> are given, labschedule assumes that <LEVEL> loops have already be processed (i.e., the first --for of the command line will be treated as <LEVEL>+1). The main benefit from using --nesting is that output of multiple commands is put in one log and output file.

Nevertheless, the switch is useful for completely different things:

- `labschedule --direct --for=*.ps ps2pdf %1 %1:r.pdf`
  call `ps2pdf` on all `*.ps` files of the current directory
- `labschedule -d -f*.jpg sh -c 'djpeg %1|pnmscale 0.5|cjpeg>sml/%1'`
  make small jpgs in a different directory (we have to add `sh -c` because we want the special characters `|` and `>` (see Section 7.1) to be interpreted by shell)
- `labschedule --direct --for='hert polzin kettner schaefer' finger %1`
  finger the explab developers

Examples:

- `labschedule --for="def1 def2 def3" --for=range(5) isp %1`
  Single host: Start the program `isp` with the three different parameters, and each command 5 times, summing to 15 executions.
- `labschedule --hosts="comp1 comp2 comp3 comp4" --for=*.stp steiner %1`
  Compute cluster: For each `*.stp` file in the current directory, start a command. Use the four mentioned hosts to perform the runs.
- `labschedule --maxtasks=20 --check=%check --for=eval(map(lambda s: s**2,range(100))) --for=range(2,11,2) calc %1 %2`
  Multiprocessor, multi-user system: Start `calc` with the first 100 square numbers, and for each square number with the five even numbers between 2 and 10 (500 executions in total). Commit maximal 20 tasks at a time, and check if there is still some idle time on the processors.
- `labschedule --for="D E F" --for=/dat/%1/*.stp --nesting=1 --labrunflag=--cvs=~/src --labrunflag=--autocvs steiner %2`
  For each of the three directories `~/dat/D`, `~/dat/E`, and `~/dat/F`, start an experiment (with the given labrun options. In each of these experiments, start `steiner` with every `*.stp` file in the current directory.

7.7 text2sus

Purpose:

Part of the ***Sus Filter Tools***: Scans a text for keywords and outputs a ***sus*** file.

Usage:

`text2sus [<option>] [<label>|<label>=<regexp>|<regexp>] ...`

Options:

`-i, --input=<FILENAME>` read from `<FILENAME>` (default: '-')

`-o, --output=<FILENAME>` write to `<FILENAME>` (default: '-')

`-n, --next=<REGEXP>` defines when to start a new record. (default: first label). For example, if you want each line to be a separate record, you can use `--next=\n`

`-b, --binary` store sus file in binary format (default: plain ascii format)

`--nolabrc` do not read `~/.labrc` and `./labrc`

With `@FILE` or `@ FILE` (some) command-line options are read from `FILE` (see section 6.2).

Defining Labels:

There are three possibilities how to define a label:

`<LABEL>` Take the first word or number after any occurrence of the `<LABEL>`.

`<LABEL>=<REGEXP>` Take the first group defined in `<REGEXP>`. If there are no parentheses in `<REGEXP>`, again take the next word or number.

`<REGEXP>` **(containing groups with labels, "`(?P<label>...)`")** Take every labeled group defined in `<REGEXP>`.

Thus,
`"label"`,
`"label=label"`,
`"label:label[:= \t]*(\w+)"`, and
`"label[:= \t]*(?P<label>w+)"`
will do the same thing. In fact the regular expression is even a bit more complicated to capture a floating point number if that seems to be the next word: `label[:= \t]*(?P<label>[+-]?\d+\.?\d*([eE][+-]?\d+)?|\w+)`

Regular Expressions: [1]

A regular expression (or REGEXP) specifies a set of strings that matches it.

---

[1] The description of regular expression is extracted and modified from Python's documentation. For more detailed information see http://www.python.org.

Regular expressions can be concatenated to form new regular expressions; if `A` and `B` are both regular expressions, then `AB` is also an regular expression. If a string `p` matches `A` and another string `q` matches `B`, the string `*pq*` will match `AB`.

A brief explanation of a part the format of regular expressions follows. For further information and a gentler presentation, consult the Regular Expression HOWTO, accessible from http://www.python.org/doc/howto/.

Regular expressions can contain both special and ordinary characters. Most ordinary characters, like '`A`', '`a`', or '`0`', are the simplest regular expressions; they simply match themselves. You can concatenate ordinary characters, so `"last"` matches the string '`last`'. (In the rest of this section, we'll write REGEXP's in `"this special style"`, usually without quotes, and strings to be matched '`in single quotes`'.)

Some characters, like '`|`' or '`(`', are special. Special characters either stand for classes of ordinary characters, or affect how the regular expressions around them are interpreted.

The special characters are:

'`.`' (Dot.) In the default mode, this matches any character except a newline.

'`^`' (Caret.) Matches the start of the string and immediately after each newline.

'`$`' Matches the end of the string and before a newline. `"foo"` matches both `'foo'` and `'foobar'`, while the regular expression `"foo$"` matches only `'foo'`.

'`*`' Causes the resulting REGEXP to match 0 or more repetitions of the preceding REGEXP, as many repetitions as are possible. `"ab*"` will match `'a'`, `'ab'`, or `'a'` followed by any number of `'b'`s.

'`+`' Causes the resulting REGEXP to match 1 or more repetitions of the preceding REGEXP. `"ab+"` will match `'a'` followed by any non-zero number of `'b'`s; it will not match just `'a'`.

'`?`' Causes the resulting REGEXP to match 0 or 1 repetitions of the preceding REGEXP. `"ab?"` will match either `'a'` or `'ab'`.

'`*?`', '`+?`', '`??`' The '`*`', '`+`', and '`?`' qualifiers are all *"greedy"*; they match as much text as possible. Sometimes this behaviour isn't desired; if the REGEXP `"<.*>"` is matched against '`<H1>title</H1>`', it will match the entire string, and not just '`<H1>`'. Adding '`?`' after the qualifier makes it perform the match in "non-greedy" or "minimal" fashion; as *few* characters as possible will be matched. Using `".*?"` in the previous expression will match only '`<H1>`'.

'`\`' Either escapes special characters (permitting you to match characters like '`*`', '`?`', and so forth), or signals a special sequence; special sequences are discussed below.

If you are not using a command-line file (see section 6.2), remember that most shells also use the backslash as an escape sequence in the command line; therefore you have to put the regular expression into `'`-quotes to prevent an interpretation by the shell.

‘[ ]’ Used to indicate a set of characters. Characters can be listed individually, or a range of characters can be indicated by giving two characters and separating them by a ‘-’. Special characters are not active inside sets. For example, "[!akm]" will match any of the characters ‘a’, ‘k’, ‘m’, or ‘!’; "[a-z]" will match any lowercase letter, and ‘[a-zA-Z0-9]’ matches any letter or digit. Character classes such as ‘\w’ or ‘\S’ (defined below) are also acceptable inside a range. If you want to include a ‘]’ or a ‘-’ inside a set, precede it with a backslash, or place it as the first character. The pattern "[]]" will match ‘]’, for example.

You can match the characters not within a range by ”complementing” the set. This is indicated by including a ‘^’ as the first character of the set; ‘^’ elsewhere will simply match the ‘^’ character. For example, "[^5]" will match any character except ‘5’.

‘|’ ‘A|B’, where A and B can be arbitrary REGEXPs, creates a regular expression that will match either A or B. This can be used inside groups (see below) as well. To match a literal ‘|’, use "\|", or enclose it inside a character class, as in "[|]".

‘(...)’ Matches whatever regular expression is inside the parentheses, and indicates the start and end of a group. To match the literals ‘(’ or ‘)’, use "\(" or "\)", or enclose them inside a character class: "[(] [)]".

‘(?...)’ This is an extension notation (a ‘?’ following a ‘(’ is not meaningful otherwise). The first character after the ‘?’ determines what the meaning and further syntax of the construct is. Extensions usually do not create a new group; "(?P<NAME>...)" is the only exception to this rule. Following are the some of the currently supported extensions.

‘(?P<NAME>...)’ Similar to regular parentheses, but the substring matched by the group is accessible via the symbolic group name NAME.

‘(?P=NAME)’ Matches whatever text was matched by the earlier group named NAME.

‘(?=...)’ Matches if "..." matches next, but doesn’t consume any of the string. This is called a lookahead assertion. For example, "Isaac (?=Asimov)" will match ‘Isaac ’ only if it’s followed by ‘Asimov’.

‘(?!...)’ Matches if "..." doesn’t match next. This is a negative lookahead assertion. For example, "Isaac (?!Asimov)" will match ‘Isaac ’ only if it’s *not* followed by ‘Asimov’.

The special sequences consist of ‘\’ and a character from the list below. If the ordinary character is not on the list, then the resulting REGEXP will match the second character. For example, "\$" matches the character ‘$’.

‘\A’ Matches only at the start of the string.

‘\b’ Matches the empty string, but only at the beginning or end of a word. A word is defined as a sequence of alphanumeric characters, so the end of a word is indicated by whitespace or a non-alphanumeric character.

‘`\B`’ Matches the empty string, but only when it is *not* at the beginning or end of a word.

‘`\d`’ Matches any decimal digit; this is equivalent to the set `"[0-9]"`.

‘`\D`’ Matches any non-digit character; this is equivalent to the set `"[^0-9]"`.

‘`\s`’ Matches any whitespace character; this is equivalent to the set `"[ \t\n\r\f\v]"`.

‘`\S`’ Matches any non-whitespace character; equivalent to the set `"[^ \t\n\r\f\v]"`.

‘`\w`’ This is equivalent to the set `"[a-zA-Z0-9_]"`, the alphanumeric characters.

‘`\W`’ This is equivalent to the set `"[^a-zA-Z0-9_]"`, the non-alphanumeric characters.

‘`\Z`’ Matches only at the end of the string.

‘`\\`’ Matches a literal backslash.

Result:

A ***sus*** file containing data from the text.

Examples:

See Section 5.2.

7.8 `table2sus`

Purpose:

Part of the ***Sus Filter Tools***: Converts a (Gnuplot-)ASCII table to ***sus*** format.

Usage:

`table2sus [<option> ... ] [<label> ...]`

If no `<label>`s are given, they are searched in the ASCII table. The number of columns is deduced from the number of space-separated words in the last uncommented line in the file. The first line with this number of words that could be labels (alphanumeric words beginning with a letter) is considered to contain the labels. If labels are not found in this way, `"col1","col2"...` are used. By default, the command acts as a filter (reads from stdin and writes to stdout).

Options:

`-i, --input=<FILENAME>` read from `<FILENAME>` (default: `'-'` for stdin)

`-o, --output=<FILENAME>` write to `<FILENAME>` (default: `'-'` for stdout)

`-s, --separator=<SEPARATOR>` use `<SEPARATOR>` as separator (default: `' '`)

`-b, --binary` store sus file in binary format (default: plain ascii format)

`-h, --help` prints this help

`--nolabrc` do not read `~/.labrc` and `./labrc`

With `@FILE` or `@ FILE` (some) command-line options are read from `FILE` (see section 6.2).

Result:

A ***sus*** file containing data from the ASCII table.

Examples:

See Section 5.2.

7.9 `sus2text`

Purpose:

Part of the ***Sus Filter Tools***: Converts ***sus*** file to a space delimited ASCII table.

Usage:

`sus2text [<option> ... ] [<label> | <label>=<expression> ...]`

If `<label>`s are given, only those `<label>`s will be processed. Otherwise all labels will be taken. By default, the command acts as a filter (reads from stdin and writes to stdout).

Options:

`-f, --filter=<EXPRESSION>` just output records where `<EXPRESSION>` is nonzero

`-s, --sort=<EXPRESSION>` sort records according to `<EXPRESSION>` (by default, take the average if there are records with the same numeric value for `<EXPRESSION>`; string values are always concatenated). Use `float(<EXPRESSION>)` to sort according to a numeric expression. Multiple `--sort` switches are possible.

`-a, --add=<LABEL=EXPRESSION>` add a new field in each record with label `<LABEL>`

`-c, --combine=[min|max|average|mean|sum|prod]` how to merge multiple entries of the same label from different records with the same `--sort <EXPRESSION>`

`-e, --eval=[strict,warn,debug,invalid]` see section 5.1 for details

`-i, --input=<SUSFILENAME>` read `<SUSFILENAME>` (default: '-' for stdin) Multiple input files are possible. The files are merged by effectively catenating them one after another. See Section 5.2.3)

`-o, --output=<FILENAME>` write to `<FILENAME>` (default: '-' for stdout)

`--nolabrc` do not read `~/.labrc` and `./labrc`

`--help-expression` print help for expressions (see section 5.1)

The switches `--add`, `--sort`, and `--filter` are processed in the same order as in the command line.

With `@FILE` or `@ FILE` (some) command-line options are read from `FILE` (see section 6.2).

Result:

A space-separated ASCII table.

Examples:

See Section 5.2.

7.10 sus2latex

Purpose:

Part of the ***Sus Filter Tools***: Converts ***sus*** file to a LaTeX table.

Usage:

`sus2latex [<option> ... ] [<label> | <label>=<expression> ...]`

If `<label>`s are given, only those `<label>`s will be processed. Otherwise all labels will be taken. By default, the command acts as a filter (reads from stdin and writes to stdout).

Options:

`-f, --filter=<EXPRESSION>` just output records where `<EXPRESSION>` is nonzero

`-s, --sort=<EXPRESSION>` sort records according to `<EXPRESSION>` (by default, take the average of numeric values if there are records with the same value for `<EXPRESSION>`; string values are always concatenated). Use `float(<EXPRESSION>)` to sort according to a numeric expression. Multiple `--sort` switches are possible.

`-a, --add=<LABEL=EXPRESSION>` add a new field in each record with label `<LABEL>`

`-c, --combine=[min|max|average|mean|sum|prod]` how to merge multiple entries of the same label from different records with the same `--sort <EXPRESSION>`

`-e, --eval=[strict,warn,debug,invalid]` see section 5.1 for details

`-i, --input=<SUSFILENAME>` read `<SUSFILENAME>` (default: '-' for stdin) Multiple input files are possible. The files are merged by effectively catenating them one after another. See Section 5.2.3)

`-o, --output=<FILENAME>` write to `<FILENAME>` (default: '-' for stdout)

`--nolabrc` do not read `~/.labrc` and `./labrc`

`--help-expression` print help for expressions (see section 5.1)

The switches `--add`, `--sort`, and `--filter` are processed in the same order as in the command line.

With `@FILE` or `@ FILE` (some) command-line options are read from `FILE` (see section 6.2).

Result:

The body of a LaTeX table (`tabular`-environment), with a simple header and footer in comment lines (see Section 5.2.9 for an example).

Examples:

See Section 5.2.

7.11 sus2plot

Purpose:

Part of the ***Sus Filter Tools***: Passes a ***sus*** file to Gnuplot.

Usage:

sus2plot [<option> ...] [<label> | <label>=<expression> ...]

If <label>s are given, only those <label>s will be processed. Otherwise all labels will be taken. By default, the command acts as a filter (reads stdin and prints stdout).

Options:

-f, --filter=<EXPRESSION> just output records that match the filter

-s, --sort=<EXPRESSION> sort records according to <EXPRESSION> (by default, take the average of numeric values if there are records with the same value for <EXPRESSION>; string values are always concatenated). In the plot, this also defines the x axis. Use float(<EXPRESSION>) to sort acording to a numeric expression. Multiple --sort switches are possible.

-a, --add=<LABEL=EXPRESSION> add a new field in each record with label <LABEL>

-c, --combine=[min,max,average,mean,sum,prod] how to merge multiple entries of the same label from different records with the same --sort <EXPRESSION>

-e, --eval=<[strict,warn,debug,invalid]> see --help-expression for details

-p, --plotcommand=<COMMAND> insert some gnuplot commands before plotting

-i, --input=<SUSFILENAME> read <SUSFILENAME> (default: '-' for stdin) Multiple input files are possible. The files are merged by effectively catenating them one after another. See Section 5.2.3)

-o, --output=<FILENAME> instead of displaying the plot, store it as <FILENAME> in one of the following formats: postscipt, eps, png or pbm (depending on the suffix of <FILENAME>).

--nolabrc do not read ~/.labrc and ./labrc

--help-expression print help for expressions (see section 5.1)

The switches --add, --sort, and --filter are processed in the same order as in the command line.

With @FILE or @ FILE (some) command-line options are read from FILE (see section 6.2).

Result:

Either a plot (in which you can navigate and zoom), or file with the plot.

Figure 1: `sus2plot` Navigation Buttons

Examples:

See Section 5.2.

7.12 sus2sus

Purpose:

Part of the ***Sus Filter Tools***: Reformats a ***sus*** file.

Usage:

`sus2sus [<option> ... ] [<label> | <label>=<expression> ...]`

If `<label>`s are given, only those `<label>`s will be processed. Otherwise all labels will be taken. By default, the command acts as a filter (reads from stdin and writes to stdout).

Options:

`-f, --filter=<EXPRESSION>` just output records where EXPRESSION is nonzero

`-s, --sort=<EXPRESSION>` sort records according to `<EXPRESSION>` (by default, take the average of numeric values if there are records with the same value for `<EXPRESSION>`; string values are always concatenated). Use `float(<EXPRESSION>)` to sort according to a numeric expression. Multiple `--sort` switches are possible.

`-a, --add=<LABEL=EXPRESSION>` add a new field in each record with label `<LABEL>`

`-c, --combine=[min,max,average,mean,sum,prod]` how to merge multiple entries of the same label from different records with the same `--sort <EXPRESSION>`

`-e, --eval=<[strict,warn,debug,invalid]>` see section 5.1 for details

`-i, --input=<SUSFILENAME>` read `<SUSFILENAME>` (default: '-' for stdin) Multiple input files are possible. The files are merged by effectively catenating them one after another. See Section 5.2.3)

`-o, --output=<FILENAME>` write to `<FILENAME>` (default: '-' for stdout)

`-b, --binary` store sus file in binary format (default: plain ascii format)

`--nolabrc` do not read `~/.labrc` and `./labrc`

`--help-expression` prints help for expressions (see section 5.1)

The switches `--add`, `--sort`, and `--filter` are processed in the same order as in the command line.

With `@FILE` or `@ FILE` (some) command-line options are read from `FILE` (see section 6.2).

Result:

A ***sus*** file.

Examples:

See Section 5.2.

8 Downloading and Contact Information

The latest release of the tool set is available for downloading from

http://explab.sourceforge.net

Here you can also report bugs or make requests for new features and learn about work in progress. We gladly accept constructive feedback, bug fixes, and improvements.

A System Environment Commands

In this appendix we list some commands that report information that may be relevant for your experiment. We use here the syntax for the `--comment` option of `labrun`. If you have other commands you believe others may find useful, please let us know so we can add them to this list.

Cache Information

IRIX:

```
Instruction cache=‘hinv‘:Instruction.cache.size[\s:]+(.*)
L1 data cache=‘hinv‘:Data.cache.size[\s:]+(.*)
```

Solaris:

```
Instruction cache=0x‘prtconf -pv‘:icache-size Bytes
L1 data cache=0x‘prtconf -pv‘:dcache-size Bytes
```

The information from `prtconf` is in hex. The `0x` before the command indicates that `labrun` should convert the number of bytes reported to decimal.

CGAL version

```
CGAL version=@$CGALROOT/include/CGAL/config.h:CGAL_VERSION\s+(.*)
```

Compiler version

```
Compiler version=‘$CC -v‘: version
```

Graphics card (IRIX)

```
Graphics card=‘hinv‘:Graphics\sboard
```

LEDA version

not using CGAL

```
LEDA version=@$LEDAROOT/CHANGES:__LEDA__
```

using CGAL

```
LEDA version=‘cgal_leda_version‘
```

Because CGAL may have been installed with a different version of LEDA than the one pointed to by `LEDAROOT`, the script `cgal_leda_version` is used to find the correct version number. The script is as follows:

```
#! /bin/sh -f

cat $CGAL_MAKEFILE |grep "/LEDA.*/incl"|sed -e s:incl:CHANGES: > /tmp/name
if [ -s /tmp/name ] ; then
  file=`cat /tmp/name`
  cat $file|grep "#define __LEDA__"|head -n 1|sed 's:#define __LEDA__::'
  rm /tmp/name
else
  echo "None"
fi
```

Notice that this script assumes the environment variable `CGAL_MAKEFILE` has been set. If this makefile does not correspond to one created with LEDA support, the value "`None`" is returned.

Qt version

```
Qt version=@$QTDIR/README:version\s+(\d+(\.\d+)*)
```

We also show here the commands used to extract the information automatically recorded in the log file in case these commands somehow correspond to different information on your system.

OS

```
OS=`uname -s`
```

Hardware

```
Hardware=`uname -m`
```

Machine

```
Machine=`uname -n`
```

Processor

IRIX:
```
Processor=`hinv | grep '.* MHZ .* Processor' |
       awk '{printf \"%d MHz %s\", $2,$4}'`
```
This command should be on a single line.

Linux:
```
Processor=@/proc/cpuinfo:cpu.MHz[:\s]+(\w)+ MHz
     @/proc/cpuinfo:model.name[\s:]+(.*)
```
This command should be on a single line.

Solaris:
```
Processor=`psrinfo -v | sed -n -e '1,4 s/.*The \(.*\)
            processor operates at \(.*\),/\2 \1/p'`
```
This command should be on a single line.

Memory size

IRIX: `Memory size=`hinv`:Main.memory.size[\s:]+(.*)`

Linux: `Memory size=@/proc/meminfo:MemTotal[\s:]+(.*)`

Solaris: `Memory size=‘prtconf -pv‘:Memory.size[\s:]+(.*)`

L2 data cache

IRIX: `L2 data cache=‘hinv‘:Secondary.*cache.size[\s:]+(.*)`

Linux: `L2 data cache=@/proc/cpuinfo:cache.size[\s:]+(.*)`

Solaris: `L2 data cache=0x‘prtconf -pv‘:ecache-size Bytes`

References

[Joh96] D. Johnson. A theoretician's guide to the experimental analysis of algorithms. http://www.research.att.com/~dsj/papers/exper.ps, 1996.

[MM99] Catherine C. McGeoch and Bernad M. E. Moret. How to present a paper on experimental work with algorithms. *SIGACT News*, 30(4):85–90, December 1999.

[MPG00] Max-Planck-Gesellschaft. Regeln zur sicherung guter wissenschaftlicher Praxis (in German). http://www.mpg.de/pri00/pri0075.htm, December 2000.

Index

I N F O R M A T I K

Below you find a list of the most recent technical reports of the Max-Planck-Institut für Informatik. They are available by anonymous ftp from `ftp.mpi-sb.mpg.de` under the directory `pub/papers/reports`. Most of the reports are also accessible via WWW using the URL `http://www.mpi-sb.mpg.de`. If you have any questions concerning ftp or WWW access, please contact `reports@mpi-sb.mpg.de`. Paper copies (which are not necessarily free of charge) can be ordered either by regular mail or by e-mail at the address below.

Max-Planck-Institut für Informatik
Library
attn. Anja Becker
Stuhlsatzenhausweg 85
66123 Saarbrücken
GERMANY
e-mail: `library@mpi-sb.mpg.de`

MPI-I-2002-4-002	F. Drago, W. Martens, K. Myszkowski, H. Seidel	Perceptual Evaluation of Tone Mapping Operators with Regard to Similarity and Preference
MPI-I-2002-4-001	M. Goesele, J. Kautz, J. Lang, H.P.A. Lensch, H. Seidel	Tutorial Notes ACM SM 02 A Framework for the Acquisition, Processing and Interactive Display of High Quality 3D Models
MPI-I-2002-2-008	W. Charatonik, J. Talbot	Atomic Set Constraints with Projection
MPI-I-2002-2-007	W. Charatonik, H. Ganzinger	Symposium on the Effectiveness of Logic in Computer Science in Honour of Moshe Vardi
MPI-I-2002-1-008	P. Sanders, J.L. Träff	The Factor Algorithm for All-to-all Communication on Clusters of SMP Nodes
MPI-I-2002-1-005	T. Polzin	?
MPI-I-2002-1-004	S. Hert, L. Kettner, G. Schäfer	?
MPI-I-2002-1-003	I. Katriel, P. Sanders, J.L. Träff	A Practical Minimum Scanning Tree Algorithm Using the Cycle Property
MPI-I-2002-1-002	F. Grandoni	Incrementally maintaining the number of l-cliques
MPI-I-2002-1-001	T. Polzin, S. Vahdati	Using (sub)graphs of small width for solving the Steiner problem
MPI-I-2001-4-005	H.P.A. Lensch, M. Goesele, H. Seidel	A Framework for the Acquisition, Processing and Interactive Display of High Quality 3D Models
MPI-I-2001-4-004	S.W. Choi, H. Seidel	Linear One-sided Stability of MAT for Weakly Injective Domain
MPI-I-2001-4-003	K. Daubert, W. Heidrich, J. Kautz, J. Dischler, H. Seidel	Efficient Light Transport Using Precomputed Visibility
MPI-I-2001-4-002	H.P.A. Lensch, J. Kautz, M. Goesele, H. Seidel	A Framework for the Acquisition, Processing, Transmission, and Interactive Display of High Quality 3D Models on the Web
MPI-I-2001-4-001	H.P.A. Lensch, J. Kautz, M. Goesele, W. Heidrich, H. Seidel	Image-Based Reconstruction of Spatially Varying Materials
MPI-I-2001-2-006	H. Nivelle, S. Schulz	Proceeding of the Second International Workshop of the Implementation of Logics
MPI-I-2001-2-005	V. Sofronie-Stokkermans	Resolution-based decision procedures for the universal theory of some classes of distributive lattices with operators
MPI-I-2001-2-004	H. de Nivelle	Translation of Resolution Proofs into Higher Order Natural Deduction using Type Theory
MPI-I-2001-2-003	S. Vorobyov	Experiments with Iterative Improvement Algorithms on Completely Unimodel Hypercubes

MPI-I-2001-2-002	P. Maier	A Set-Theoretic Framework for Assume-Guarantee Reasoning
MPI-I-2001-2-001	U. Waldmann	Superposition and Chaining for Totally Ordered Divisible Abelian Groups
MPI-I-2001-1-007	T. Polzin, S. Vahdati	Extending Reduction Techniques for the Steiner Tree Problem: A Combination of Alternative-and Bound-Based Approaches
MPI-I-2001-1-006	T. Polzin, S. Vahdati	Partitioning Techniques for the Steiner Problem
MPI-I-2001-1-005	T. Polzin, S. Vahdati	On Steiner Trees and Minimum Spanning Trees in Hypergraphs
MPI-I-2001-1-004	S. Hert, M. Hoffmann, L. Kettner, S. Pion, M. Seel	An Adaptable and Extensible Geometry Kernel
MPI-I-2001-1-003	M. Seel	Implementation of Planar Nef Polyhedra
MPI-I-2001-1-002	U. Meyer	Directed Single-Source Shortest-Paths in Linear Average-Case Time
MPI-I-2001-1-001	P. Krysta	Approximating Minimum Size 1,2-Connected Networks
MPI-I-2000-4-003	S.W. Choi, H. Seidel	Hyperbolic Hausdorff Distance for Medial Axis Transform
MPI-I-2000-4-002	L.P. Kobbelt, S. Bischoff, K. Kähler, R. Schneider, M. Botsch, C. Rössl, J. Vorsatz	Geometric Modeling Based on Polygonal Meshes
MPI-I-2000-4-001	J. Kautz, W. Heidrich, K. Daubert	Bump Map Shadows for OpenGL Rendering
MPI-I-2000-2-001	F. Eisenbrand	Short Vectors of Planar Lattices Via Continued Fractions
MPI-I-2000-1-005	M. Seel, K. Mehlhorn	Infimaximal Frames: A Technique for Making Lines Look Like Segments
MPI-I-2000-1-004	K. Mehlhorn, S. Schirra	Generalized and improved constructive separation bound for real algebraic expressions
MPI-I-2000-1-003	P. Fatourou	Low-Contention Depth-First Scheduling of Parallel Computations with Synchronization Variables
MPI-I-2000-1-002	R. Beier, J. Sibeyn	A Powerful Heuristic for Telephone Gossiping
MPI-I-2000-1-001	E. Althaus, O. Kohlbacher, H. Lenhof, P. Müller	A branch and cut algorithm for the optimal solution of the side-chain placement problem
MPI-I-1999-4-001	J. Haber, H. Seidel	A Framework for Evaluating the Quality of Lossy Image Compression
MPI-I-1999-3-005	T.A. Henzinger, J. Raskin, P. Schobbens	Axioms for Real-Time Logics
MPI-I-1999-3-004	J. Raskin, P. Schobbens	Proving a conjecture of Andreka on temporal logic
MPI-I-1999-3-003	T.A. Henzinger, J. Raskin, P. Schobbens	Fully Decidable Logics, Automata and Classical Theories for Defining Regular Real-Time Languages
MPI-I-1999-3-002	J. Raskin, P. Schobbens	The Logic of Event Clocks
MPI-I-1999-3-001	S. Vorobyov	New Lower Bounds for the Expressiveness and the Higher-Order Matching Problem in the Simply Typed Lambda Calculus
MPI-I-1999-2-008	A. Bockmayr, F. Eisenbrand	Cutting Planes and the Elementary Closure in Fixed Dimension
MPI-I-1999-2-007	G. Delzanno, J. Raskin	Symbolic Representation of Upward-closed Sets
MPI-I-1999-2-006	A. Nonnengart	A Deductive Model Checking Approach for Hybrid Systems
MPI-I-1999-2-005	J. Wu	Symmetries in Logic Programs
MPI-I-1999-2-004	V. Cortier, H. Ganzinger, F. Jacquemard, M. Veanes	Decidable fragments of simultaneous rigid reachability
MPI-I-1999-2-003	U. Waldmann	Cancellative Superposition Decides the Theory of Divisible Torsion-Free Abelian Groups
MPI-I-1999-2-001	W. Charatonik	Automata on DAG Representations of Finite Trees
MPI-I-1999-1-007	C. Burnikel, K. Mehlhorn, M. Seel	A simple way to recognize a correct Voronoi diagram of line segments
MPI-I-1999-1-006	M. Nissen	Integration of Graph Iterators into LEDA

MPI-I-1999-1-005	J.F. Sibeyn	Ultimate Parallel List Ranking ?
MPI-I-1999-1-004	M. Nissen, K. Weihe	How generic language extensions enable "open-world" desing in Java
MPI-I-1999-1-003	P. Sanders, S. Egner, J. Korst	Fast Concurrent Access to Parallel Disks