ExpLab
A *Tool Set for Computational Experiments*
— A Short Tutorial —

http://explab.sourceforge.net/

Susan Hert Lutz Kettner Tobias Polzin Guido Schäfer

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

November 18, 2002

Abstract

The *Tool Set for Computational Experiments* is intended to **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.

We first discuss the installation of the tool set (Section 1) and then show through examples how the tools might be used to

1. run computational experiments such that they can be reproduced at a later time (Section 2),
2. convert (certain parts of) the raw data output by the experiment to several more sophisticated data formats such as LaTeX table, gnuplot, etc. (Section 3).

The *Tool Set for Computational Experiments* is available at
http://explab.sourceforge.net

Contents

1 Installation

To install the *Tool Set for Computational Experiments* do the following:

1. Unpack the `ExpLab-<vers>.tar.gz` file
   `tar xvzf ExpLab-<vers>.tar.gz`
   This will create a directory `exp` in your current directory. Subsequently, we use `<dir>` to refer to the current directory.
2. Change the paths in the `Makefile`
3. Make sure that `python` is your search path and that `/usr/bin/env` exists. If not, change the line at the top of each script to correspond to the place where `python` can be found on your system.
4. Run `make install`.

You might want to have a look at a few examples illustrating what the output processing tools can do:

```
cd <dir>/exp/example
sus-tools-demo
```

2 Making Experiments Reproducible

A typical scenario in which the *Tool Set for Computational Experiments* might be of great use is that we have a program that performs certain experiments and we want to be able to repeat or at least recapitulate later (possibly after several years) how these experiments were done.

Generally, the following two steps are necessary to make an experiment reproducible.

1. All source files that are necessary to perform the experiment are archived.
2. Certain parameters used for the compilation and execution of the experiment are logged.

The easiest way to do this is to use the *Tool Set for Computational Experiments* in combination with a version control system such as the Concurrent Version System (CVS) (available at http://www.cvshome.org/). Using CVS enables us to restore all source files of a particular experiment if we know, for example, the date on which the experiment was run. The bookkeeping mechanisms provided by the tool set (in particular, the program `labrun`) record the dates of experiments as well as many other relevant environmental data, which allows one later to rerun the experiment (provided that the same environment, *i.e.*, machine, compiler, etc., is still available). Rerunning is easily accomplished using the `labrerun` tool.

2.1 Putting the Experiment under CVS Control

Assume all files that are necessary to run the experiment are located in a directory `<exp_dir>`. We put this directory under CVS control first. CVS is used to record the history of all source files in `<exp_dir>`. It stores a copy of each file in a repository and keeps a history of the changes made. We briefly review how to set up a CVS repository and how to commit all source files to it.

2.1.1 Setting Up a CVS Repository

If you have never used CVS before, you will have to set up a CVS repository first.

1. Make sure that CVS is properly installed in your system. Set the environment variable `CVSROOT` to the directory in which you want to install a CVS repository. For example, if you are using bach, include the following line in your `.bashrc`
   ```
   export CVSROOT=$HOME/cvs
   ```
2. After that,
   ```
   cvs init
   ```
   will create a CVS repository in `$CVSROOT`

2.1.2 Importing Source Files to the CVS Repository

Next, you will have to import all source files of the experiment into the repository. Do the following:

1. Change to the experiment's directory
   ```
   cd <exp_dir>
   ```
2. Put all files into the CVS repository
   ```
   cvs import -m "<comment>" <name> <vendor_tag> <release_tag>
   ```
   All arguments are required. A comment should be given as `<comment>`. `<name>` specifies the name under which all files in `<exp_dir>` are stored in the CVS repository (which is `$CVSROOT/<name>`). `<vendor_tag>` and `<release_tag>` are required but are of no meaning in the current context, so they may be set arbitrarily. For example, the above statement might look like this
   ```
   cvs import -m "imported sources" my_experiment exp start
   ```
3. Verify that everything went fine and use the directory under CVS control to perform your experiments:
   ```
   cd ..
   mv <exp_dir> <exp_dir>.orig
   ```
   Get a local copy of the just imported files
   ```
   cvs checkout <name>
   ```
   This should produce a new directory `<name>` containing identical copies of all source files in `<exp_dir>.orig`. Verify that both directories are really the same
   ```
   diff -f <exp_dir>.orig <name>
   ```
   If no output was produced by the last command you can safely remove the original directory
   ```
   rm -rf <exp_dir>.orig
   ```
   Now `<name>` contains a valid version of your experiment, which is under CVS control. Use this from now on.

2.1.3 Basic Commands of CVS

We list some more basic commands. Please refer to, *e.g.*, `http://www.cvshome.org/` for a more detailed description.

- Retrieving a local copy of a specified directory `<name>` under CVS
  `cvs checkout <name>`
- Updating the local copy (the whole directory or a single file `<file>`) from the CVS repository
  `cvs update` or `cvs update <file>`
- Committing changed files (all changed files or a single file `<file>`) to the CVS repository
  `cvs commit -m "<comment>"` or
  `cvs commit -m "<comment>" <file>`
- Adding a (new) directory or file `<file>` to the CVS repository
  ```
  cvs add <file>
  cvs commit -m "<comment>" <file>
  ```
- Removing a directory or file `<file>` from the CVS repository
  ```
  cvs remove <file>
  cvs commit -m "<comment>" <file>
  ```
- Reading the log of a file under CVS
  `cvs log <file>`

⇒ **Tutorial 1:** In the directory `exp/examples/sorting` you will find a `Makefile` and a C source file `sort-demo.c`. Put these source files under CVS control.

Solution: Assuming that a CVS repository was set up:

```
cd exp/examples/sorting
cvs import -m "sorting sources imported" sorting sort start
```

Next, get a local copy of the imported sources (here in `~/sorting`):

```
cd
cvs checkout sorting
```

2.2 Compilation and Execution

2.2.1 The `labmex` Tool

The *Tool Set for Computational Experiments* contains a program `labmex` that facilitates the compilation and execution of a program. In general, the syntax is as follows:

```
labmex [<options>] <executable> [<arguments>]
```

`<executable>` refers to the program being compiled and executed. `<arguments>` specify the arguments given to `<executable>` (if any). `<options>` may be used to give different options to `make`. By default, `labmex` uses the command `make` to build the specified target `<executable>` and then, after successful compilation, executes it. If the compilation step fails, `labmex` will produce a compilation log file `<executable>-<date>-<time>.clog`. Type `labmex --help` for more help.

⇒ **Tutorial 2:** Use `labmex` to compile and execute `sort-demo`. Find out which arguments `sort-demo` requires and perform a sorting experiment on 5000 numbers and 10 random sequences.

Solution:
```
cd sorting
labmex ./sort-demo
... find out proper usage ...
labmex ./sort-demo 5000 10
```

The solution to this and many other of the tutorial tasks can be shown with the program `create-tutorial` in the tutorial's directory. Running this program creates several `tut<tutorial-nr>` files.

⇒ **Tutorial 3:** Next, play around with different options: (1) Force a `make clean` before the compilation of `sort-demo`. (2) Force the `.clog` file to be created even if compilation is successful. (3) Use a different compiler, *e.g.*, `gcc`, by redefining the `CC` variable and set the makefile variable `LFLAGS` to the optimization level `-O3`. (4) Combine (1)–(3) by writing an option file and call `labmex` on this option file.

Solution:
```
(1) labmex -c before ./sort-demo 5000 10
(2) labmex -k ./sort-demo 5000 10
(3) labmex -m CC=gcc -m LFLAGS=-O3 ./sort-demo 5000 10
```
(4) Put all options into a file, say `labmex.opt1`, and call
```
    labmex @ labmex.opt1 ./sort-demo 5000 10
```

2.2.2 The labrun Tool

The program `labrun` provides a means to run experiments and automatically log the context of the experiments.

```
labrun [<options>] <executable> [<arguments>]
```

The program `<executable>` should run non-interactively. `labrun` will first check whether the provided directories (the default is just the current directory) is under CVS and up to date and then, if so, executes the program `<executable>` with the given arguments `<arguments>` (if any). `<options>` might be used to control the behaviour of `labrun`. Type `labrun --help` to get more help.

In general, `labrun` generates at least two files: a `.out` file that contains the output of the experiment and a `.log` file that contains certain environment information. Additionally, if the program produces some error messages (written to the standard error stream), these will be recorded in a `.err` file. All file names are of the form

```
<executable>-<date>-<time>.[out|log|err]
```

and can be found in a subdirectory `./lab_log`. A link `current.[out|log|err]` will point to the files for the latest experiment performed.

In the following tutorials, we illustrate how various options of `labrun` might be used in conjunction with `labmex`.

⇒ **Tutorial 4:** Call `labrun` on `labmex` to compile and execute `sort-demo` with $n = 5000$ and $r = 10$. Then, check the output file `current.out` and the log file `current.log` in `./lab_log`.

Solution:

```
labrun labmex ./sort-demo 5000 10
cd lab_log
```

. . . look at the `.log` and `.out` files . . .

⇒ **Tutorial 5:** Find out which option is used to let `labrun` automatically commit and update the CVS repository if necessary.

Solution:

```
labrun -a labmex ./sort-demo 5000 10
```

⇒ **Tutorial 6:** Write an option file `labrun.opt1` for `labrun` such that the call

```
labrun @ labrun.opt1 labmex ./sort-demo 5000 10
```

does the following: (1) The log files are put into a directory `./res`. (2) The tag `5000-10` is used instead of the `<date>-<time>` stamp. (3) The experiment is executed in batch mode. (4) All data are stored in only one log file.

Solution: The corresponding file `labrun.opt1` would look like this:

`labrun.opt1`

```
--log=./res
--tag=5000-10
--batch
--one
```

⇒ **Tutorial 7:** Use the option `--comment` to add your own comments to the log file. (Type `labrun --help-comment` to get help.) Try to record the value of some environment variables, *e.g.*, `CXXFLAGS`, in the log file.

Solution: The following option file will add a table detailing the free and allocated dynamic memory on the system (result of `free`) and the version of `gcc` (result of `gcc -v`) as a comment to the log file. Moreover, it records the value of the environment variable `CXXFLAGS`.

`labrun.opt2`

```
--comment=Memory=‘free‘
--comment=Compiler=‘gcc -v‘
--env=CXXFLAGS
```

Several experiments might be started by calling `labrun` several times. However, it is more convenient to put all calls into one file and to separate different experiments via the keyword `NEX`.

⇒ **Tutorial 8:** Write an option file `labrun.opt3` such that the call

```
labrun @ labrun.opt3
```

performs the following set of experiments: (1a) `sort-demo` with $n = 5000$ and $r = 10$. (1b) `sort-demo` is compiled with the additional optimization flag `-O3` and run with $n = 5000$ and $r = 10$. (2a) `sort-demo` with $n = 10000$ and $r = 10$. (2b) `sort-demo` is compiled with the additional optimization flag `-O3` and run with $n = 10000$ and $r = 10$. Make sure that each call does indeed recompile the source code. Redefine the `<date>-<time>` tag such that it encodes each experiment's arguments, *e.g.*, you might use `10000-10-O3` for (2b). The log files of all experiments should keep track of the version number of the compiler used. Additionally, each experiment should be launched in verbose mode. All experiments should run in batch mode.

Solution:

labrun.opt3

```
# experiment (1a)
--comment=Compiler=‘gcc -v‘
--tag=5000-10-no-O
--verbose
--batch
labmex -c before ./sort-demo 5000 10

NEX

# experiment (1b)
--comment=Compiler=‘gcc -v‘
--tag=5000-10-O3
--verbose
--batch
labmex -c before -m LFLAGS=-O3 ./sort-demo 5000 10

NEX

# experiment (2a)
--comment=Compiler=‘gcc -v‘
--tag=10000-10-no-O
--verbose
--batch
labmex -c before ./sort-demo 10000 10

NEX

# experiment (2b)
--comment=Compiler=‘gcc -v‘
--tag=10000-10-O3
--verbose
--batch
labmex -c before -m LFLAGS=-O3 ./sort-demo 10000 10
```

Each of the above four experiments uses the three flags `--comment=Compiler=‘cc-v’`, `--verbose`, and `--batch`. If all your experiments are meant to be run with a common set of command-line options, a global resource file can be created that contains these common options. The program `labsetup` is available to help you create this resource file. The options recorded in this file will be read will `labrun` is executed and added to the command-line arguments. In the case of conflicts, command-line arguments take precendence over those recorded in a resource file. For options that may appear more than once on the command line, the union of the command-line and resource file options is used.

⇒ **Tutorial 9:** Run the program `labsetup` to create a global resource file that indicates that `labrun` should always run in verbose mode and in batch mode and should always record the version of the compiler `cc`.

Solution:

labsetup session

```
[...]

labsetup: Settings for labrun

Execute labrun in verbose mode? [n]
y

[...]

Run in the background? [n]
y

[...]

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='gcc -v'

[...]
```

⇒ **Tutorial 10:** Now that a global resource file has been created, modify the option file created for Tutorial 8 by removing the common options of the four experiments and rerun this experiment, putting the resulting log files in a different directory, say `lab_log2`. Examine the log files from the two sets of experiments to see that the same information is recorded in both.

Solution:

```
labrun.opt4
# experiment (1a)
--tag=5000-10-no-O
--log=lab_log2
labmex -c before ./sort-demo 5000 10

NEX

# experiment (1b)
--tag=5000-10-O3
--log=lab_log2
labmex -c before -m LFLAGS=-O3 ./sort-demo 5000 10

NEX

# experiment (2a)
--tag=10000-10-no-O
--log=lab_log2
labmex -c before ./sort-demo 10000 10
```

```
NEX

# experiment (2b)
--tag=10000-10-O3
--log=lab_log2
labmex -c before -m LFLAGS=-O3 ./sort-demo 10000 10
```

2.3 Rerunning an Experiment: The `labrerun` Tool

Once an experiment has been performed using `labrun` it is easy to rerun the same experiment later.

```
labrerun [<options>] <log_file>
```

In its simplest form, `labrerun` takes a log file `<log_file>` and then repeats the experiment specified in this log file. `labrerun` will first check out the sources that were used when the experiment was performed. Then, it will reconstruct all parameters that are needed by a call of `labrun` in order to perform the same experiment. The environment variables will be set as recorded in `<log_file>`. New `.[out|log|err]` files for the rerun will be generated by default in the directory containing the original log file. Several options are provided by `labrerun`; many of them are the same as for `labrun`.

⇒ **Tutorial 11:** Rerun the experiment (2b) from Tutuorial 8. (If you like, you can even modify the source code of `sort-demo`, perform a new set of expriments and after that repeat the experiment (2b).)

Solution:

```
labrerun ./lab_log/sort-demo-10000-10-O3.log
```

Sometimes it might be desirable to keep the CVS sources that are checked out by `labrerun`. (By default, the latest versions of the sources are retrieved from CVS after `labrerun`.) This can be achieved by using the option `--keep`. The `--nocvs` option specifies that the current source files should be used instead of those specified in the log file. In cases where one wants to disregard certain enviornment variable definitions given in `<log_file>`, the `--ignore` option comes in handy.

⇒ **Tutorial 12:** (1) Rerun experiment (2b) from Tutuorial 8 but keep the sources that are checked out. (2) Rerun experiment (1a) from Tutorial 8 but use the current sources. (3) Restore the latest version of the source code in the directory.

Solution:

```
labrerun --keep ./lab_log/sort-demo-10000-10-O3.log
labrerun --nocvs ./lab_log/sort-demo-5000-10-no-O.log
cvs up -A
```

2.4 Scheduling Multiple Experiments: The `labschedule` Tool

The program `labschedule` is available to help in the running of multiple experiments either sequentially or in parallel.

```
labschedule [<options>] <executable> [<arguments>]
```

The `--for` option of `labschedule` allows you to specify a set of for loops, each nested inside the other in the order given on the command line. The innermost command is a call to `labrun` to run `<executable>` with the given `<arguments>`. A variable is defined for each loop. The name of a loop's variable is simply the number of the loop preceded by a % (*e.g.*, %1). The occurrence of a variable in the command line will be replaced by the individual values of the loop range. Loop range values can be given in a number of ways. The simplest way is to simply specify the individual values on the command line.

⇒ **Tutorial 13:** Use `labschedule` to execute four calls to `labrun` that will run four experiments to sort $n = 500$, 1000, 1500 and 2000 numbers with $r = 10$.

Solution:

```
labschedule --for '500 1000 1500 2000' sort-demo %1 10
```

Each call the `labschedule` creates a log file that records information about the tasks that were scheduled, including when they were started, when they finished, and the command used for scheduling. This file can be found in `./lab_log`, along with a `.out` file containing the output of every successfully completed experiment and a `.err` file containing the output of every failed experiment. The file names have a form similar to those created by `labrun`:

```
<executable>-<date>-<time>.[out|log|err]
```

where `<executable>` is `schedule` by default. The calls to `labrun` issued by `labschedule` use the option `--name=schedule-%1-%2-`··· to indicate that `schedule` and the names of the loop variables should be used to name the output files of `labrun`.

⇒ **Tutorial 14:** List the log and output files created by Tutorial 13

Solution:

```
ls ./lab_log/schedule*
```

A loop range can also be specified using any python expression. Particularly helpful here is the `range` expression. When given one integer argument x the range expands to $0, \ldots, x-1$. Two arguments may be used to specify a starting point other than 0 and a third argument can specify a step size other than 1.

⇒ **Tutorial 15:** Reschedule the experiments from Tutorial 13 using the `range` command to specify the loop values. Run `labschedule` in verbose mode to see what happens.

Solution:

```
labschedule -v --for 'range(500, 2001, 500)' sort-demo %1 10
```

You will notice that the expriments were actually not rerun (unless something went wrong in Tutorial 13). The `labschedule` tool recognizes that the specified experiments have already successfully completed and, by default, it doesn't run a successful experiment again. This behavior can be changed by using the `--noskip` option.

⇒ **Tutorial 16:** Rerun the experiments from Tutorial 13 and then examine the `./lab_log` directory to find the files created by the rerun.

Solution:

```
labschedule -v --noskip --for 'range(500, 2001, 500)' sort-demo %1 10
ls ./lab_log/schedule*
```

Another alternative for rerunning the experiments would be to change the name associated with the experiment using the `--name` option.

The same syntax used for providing values of comments in the log file (Tutorial 8) is also available for specifying `for` loop ranges. In particular, the values can be read from a file by specifying the name of the file after a `@` on the command line.

⇒ **Tutorial 17:** Run five experiments with $n = 600$, 1000, 1250, 1500, and 2000 and $r = 10$.

Solution:

The following input file provides 5 values for n.

n_values

```
600 1000 1250 1500 2000
```

```
labschedule --for @n_values sort-demo %1 10
```

⇒ **Tutorial 18:** Add a second loop that varies the value of r from 10 to 14, stepping by 1, as well as varying n as in the previous tutorial. Use the `--print` option to simply print the commands that would be executed without actually running them.

Solution:

```
labschedule --print --for @n_values --for 'range(10, 15)' sort-demo %1 %2
```

Without using the `--print` option in Tutorial 18, 20 calls to `labrun` would have been made, resulting in 20 separate `labrun` log files and output files, which is a lot. You can use the `--nesting` option to reduce the number of files created. This option indicates how deeply nested the calls to `labrun` produced by `labschedule` should be. If the nesting level is set to something less than the number of loops specified on the command line, the command that `labschedule` will give to `labrun` to execute will be another call to `labschedule`. In this second call to `labschedule`, the `--direct` flag is used to indicate that the second `labschedule` should run the executable it is given directly, without another call to `labrun`.

⇒ **Tutorial 19:** Use the `--nesting` option to run the experiments from Tutorial 18 with only five calls to `labrun`. Change the name prefix for the experiment to `sort` using the `--name` option.

Solution:

```
labschedule --name=sort --nesting=1 --for @n_value --for 'range(10, 15)'
     sort-demo %1 %2
```

The `--maxtasks` option of `labschedule` allows you to specify that more than one task can be started at once. This generally makes sense only for multiprocessor systems.

⇒ **Tutorial 20:** Execute the following 4 `sort-demo` experiments simultaneously: $n = 5000$, $r = 10$; $n = 10000$, $r = 10$; $n = 5000$; $r = 5$; $n = 10000$, $r = 5$.

Solution:

```
labschedule --max=4 --for '5000 10000' --for '5 10' sort-demo %1 %2
```

3 Data Conversion

The *Tool Set for Computational Experiments* provides some output filtering tools that should help to analyze the raw data output of an experiment and to facilitate the conversion between different formats.

To achieve maximum flexibility, these tools provide two kind of filters: filters that convert text given in any form to an internal *sus* data format and filters that convert a *sus* file to some other formats, such as an ASCII table, a LaTeX table, a `gnuplot` data or graph. For example, the filters `table2sus` and `text2sus` belong to the former kind and `sus2text`, `sus2latex`, `sus2plot` belong to the latter kind.

All filters expect input from standard input and write output to standard output by default. It is also possible to use the `--input` or `--output` switches to specify the files explicitly. If there is no such switch, a filter program `<filter_program>` should be called using redirection:

```
<filter_program>  <  <infile>  >  <outfile>
```

The *sus* file is a text file following Python's syntax (see `http://www.python.org`).

3.1 Conversion to *sus* Files

3.1.1 `text2sus`

`text2sus` is a very powerful filter. In general, the syntax is as follows:

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

It scans an input file and extracts strings following specified keywords `<label>` or strings that match user-defined regular expressions `<regexp>`. Next, we will discuss some of the features provided by `text2sus`. For more detailed information please refer to the manual.

Subsequently, we consider an output file, say `out.txt`, produced by `sort-demo` (only the relevant part is shown):

`out.txt [...]`

```
STARTING experiment...
* [1] ascending ordered sequence <1,...,n>
  type = ascending
    T0 =     0.16
    T1 =     0.00
    T2 =     0.16
    T3 =     0.09
     r =        1

* [2] descending ordered sequence <n,...,1>
  type = descending
    T0 =     0.16
    T1 =     0.22
    T2 =     0.40
    T3 =     0.10
     r =        1

* [3] r sequences of n random numbers from [1,...,n]
  type = random
    T0 =     0.15
    T1 =     0.16
    T2 =     0.36
    T3 =     0.00
     r =       10

ready!
```

The simplest way of extracting certain keyword-value pairs from some text file is by specifying a keyword `<label>`. The first number or word after any occurrence of `<label>` will be interpreted as a value.

⇒ **Tutorial 21:** Convert the output file `out.txt` into a *sus* file and try to extract the `type` data and the running times of the insertion sort algorithm (`T1`)

Solution: `text2sus type T1 < out.txt`

⇒ **Tutorial 22:** Write an option file `text2sus.opt1` such that the following call

```
text2sus @ text2sus.opt1 < out.txt
```

produces a *sus* file that contains the running times of all four algorithms together with the type of the experiment and a short description of the experiment. Use the `sus2text` filter to verify your result, *i.e.*,

```
text2sus @ text2sus.opt1 < out.txt | sus2text
```

should produce something similar to this:

```
                              result
                                      description        type   T0   T1   T2   T3

          ascending ordered sequence <1,...,n>  ascending 0.16 0.00 0.16 0.09
         descending ordered sequence <n,...,1> descending 0.16 0.22 0.40 0.10
r sequences of n random numbers from [1,...,n]     random 0.15 0.16 0.36 0.00
```

Solution:

```
                           text2sus.opt1
description=\* \[[123]\] (.*)
type
T0
T1
T2
T3
```

Here, for the description, we need to define a regular expression: '`\* \[[123]\] (.*)`'. The '`\* \[`' matches the '`* [`' at the beginning of the lines we are interested in, '`[123]`' matches '1', '2' or '3', '`\]`' matches the '`]`', and finally, '`(.*)`' matches the rest of the line. As '`(.*)`' is the first group in the regular expression, the value of this match will be stored with label `description`.