Appendix G

Concurrent Versions System (CVS)

A Quick Introduction to CVS

*Wolfgang Dobler**

Revision: 1.25 , Date: 2005/11/28 22:32:29

Contents

*Please send comments etc. to Wolfgang.Dobler@ucalgary.ca

1 What CVS does

CVS (Concurrent Versions System)

- is a system that lets groups of people work simultaneously on groups of files (for instance a numerical code, a LaTeXpaper, a set of HTML pages, etc.)
- allows you to retrieve older versions of (the important files in) a directory tree, identified by date, specific tags, ...
- forces you (to some extent) to write log messages for any changes you make to your code. These messages are recorded and with '*cvs log ...*' you can obtain a full annotated changelog history for a given file or module
- can also be used to keep your computing environment in sync (e.g. '*~/bin*', '*~/idl/lib*', '*~/tex/include*') across different computers

2 Nomenclature

Some specific terms you should know:

repository: The directory structure where CVS stores the files it manages, together with some administrative files

module: Essentially, a directory tree subject to version control. More formally, a module is a directory tree listed in '*CVSROOT/modules*', which can be accessed under the module name instead of the full path

tags, rtags: Labels attached to the files (possibly directories) and modules, allowing to identify them more easily

revision: A numerical or alpha-numerical tag identifying the version of a file

check in (commit) / check out: Write your modified version of a file/module to the repository (commit); retrieve the latest or a particularly specified version from the repository (check out)

3 Getting help

There are several levels of information available for the CVS commands.

1. '`cvs -H` *command*' or '`cvs --help` *command*' gives an overview of *command*. A (brief) overview of the '`cvs`' command itself is obtained by '`cvs -H`'.
2. The CVS manpage ('`man cvs`') shows a brief overview over all CVS commands, followed by a detailed list of general options and a more detailed description of the individual commands.
3. `http://www.cvshome.org/` is the standard reference site for CVS. Apparently, three lists of frequently asked questions are available, two of which (`http://www.loria.fr/~molli/cvs/cvs-FAQ-1.4/cvsfaq0.html` and `http://www.loria.fr/`

cgi-bin/molli/fom.cgi) are quite extensive, but not up to date, while the third (href="http://ccvs.cvshome.org/fom/fom.cgi") is up to date and apparently quite short.

4. The Cederqvist manual ('Version Management with CVS' by Per Cederqvist et al.) is the official (and comprehensive) documentation to CVS. You can read it online as *info* file with 'info cvs' (or using *Emacs* as info reader), read the HTML version under 'http://www.cvshome.org/docs/manual/cvs.html (Debian GNU/Linux also installs it under '*/usr/doc/cvs/html-info/cvs_toc.html*'), or get a PostScript version from the web sites mentioned above.
5. A good book on CVS, which is freely available online is 'Open Source Development with CVS' by Karl Fogel (http://cvsbook.red-bean.com/).

4 Environment variables

CVSROOT points to the repository you want to use. If you use a local repository, CVSROOT simply contains the file name of the top CVS directory. For server/client access, the CVSROOT variable has the form ':pserver:*user@server*:*directory*'; see Section 5 for an example.

CVSEDITOR determines the editor used for the log messages you have to enter. Set this to 'vi', 'emacsclient' or whatever you like, if you are not happy with the editor specified in $EDITOR for that purpose.

5 A sample session

> ***Note:*** *All examples below assume you are using the server/client method to access the repository. If this is not the case, you need to set CVSROOT accordingly and just ignore the cvs login and cvs logout commands.*
>
> *All examples assume that some directories and modules (like test) have already been checked in; this is because this document was written for a specific group working with a specific code. To really start from scratch you may want to have a look at other documentation.*

Set the CVSROOT environment variable to ':pserver:$USER@cvsserver.somehwere.net:/home/cvs/cvsroot', where $USER should be the user name on the CVS server. You need to adapt the server name (cvsserver.somehwere.net in the example) and the repository path (our example /home/cvs/cvsroot corresponds to a system where a user '*cvs*' owns the repository).

Log in for server/client mode:

```
unix> cvs login
(Logging in to USER@cvsserver.somehwere.net)
CVS password: ........
```

Get a working copy of module *test*:

unix> `cd ~/f90/work`
work> `cvs checkout test` (or, synonymically, `cvs co test`)
[lengthy output]

This gets you the latest version of module 'test' from the repository; it creates a directory '*test/*'.

work> `cd test/`

Edit the files you want to modify:

work/test> `[vi/emacs] src/run.f90`
work/test> `[vi/emacs] runs/run1/run.in`

Maybe you also want to delete a file and create a new one:

work/test> `rm unnecessary.txt; cvs remove unnecessary.txt`
work/test> `cp src/start.f90 src/start_test.f90`
work/test> `[vi/emacs] src/start_test.f90; cvs add src/start_test.f90`

Note that `cvs add/remove` does not change anything in the repository before you also commit the changes:

work/test> `cvs update` (get new version from server if available)

```
R unnecessary.txt
cvs server: Updating idl
cvs server: Updating runs
cvs server: Updating runs/run1
M runs/run1/run.in
cvs server: Updating src
A src/start_test.f90
M src/run.f90
```

(your output will look different). This indicates that the files *runs/run1/run.in* and *src/run.f90* have been **m**odified by you, while *src/start_test.f90* has been **a**dded and *unnecessary.txt* **r**emoved. Now commit the changes:

work/test> `cvs commit`

```
cvs commit: Examining .
cvs commit: Examining idl
cvs commit: Examining runs
cvs commit: Examining runs/run1
cvs commit: Examining src
Removing unnecessary.txt;
/home/cvs/cvsroot/test/unnecessary.txt,v  <--  unnecessary.txt
new revision: delete; previous revision: 1.1.1.1
done
Checking in runs/run1/run.in;
/home/cvs/cvsroot/test/runs/run1/run.in,v  <--  run.in
new revision: 1.2; previous revision: 1.1
done
RCS file: /home/cvs/cvsroot/test/src/start_test.f90,v
Checking in src/start_test.f90;
```

```
/home/cvs/cvsroot/test/src/start_test.f90,v  <--  deriv_6th.f90
initial revision: 1.1
done
Checking in src/run.f90;
/home/cvs/cvsroot/test/src/run.f90,v  <--  run.f90
new revision: 1.2; previous revision: 1.1
done
```

Note:

1. *You can choose which files or directories/modules to commit:*

   work> `cvs commit test` *(equivalent to the above)*
   work> `cvs commit test/src/run.f90` *(commit just one file)*

2. *If you do not want an editor to be started each time you commit, you can issue the log message directly when committing (option '`-m`'):*

   work> `cvs commit -m "Fixed entropy diffusion" test/src/run.f90`

To get information about the changes that *run.csh* has gone through, use

work/test> `cvs log src/run.f90`
[lengthy output]

If you want to know what the differences are between your working version of the code and the version you were starting with, type

work/test> `cvs diff src/run.f90`
[no output]

Therefore, your version has not been modified since the last *update* or *commit*. By contrast, to see the differences with respect to the latest version in the repository, use '`cvs diff -r HEAD run.csh`'.

But you can also check what made revision 2.0 so different from revision 1.1:

work/test> `cvs diff -r1.1 -r2.0 src/run.f90`
[output in Unix diff(1) format]

The command '`cvs status`' shows you the current status of a file/directory or repository:

work/test> `cvs status src/run.f90`

```
===================================================================
File: run.f90           Status: Up-to-date

   Working revision:    2.0
   Repository revision: 2.0     /home/cvs/cvsroot/test/src/run.f90,v
   Sticky Tag:          2.0
   Sticky Date:         (none)
   Sticky Options:      (none)
```

When you are done with the code, you can check whether you have committed all your changes:

```
work/test>  cd ..   (You must be immediately above
the directory you were working on)
work>  cvs release test
M start.csh
You have [1] altered files in this repository.
Are you sure you want to release directory 'test': n
** 'release' aborted by user choice.
```

In this example you had not, and entered 'n' to cancel the release. Now commit the modified file *start.csh* and release again:

```
work>  cvs commit -m "Set nwidth to 17" test/start.csh
work>  cvs release -d test   (Be careful when using the '-d' option!)
You have [0] altered files in this repository.
Are you sure you want to release (and delete) directory 'test': y
```

With the '-d' option, *release* removes the working copy of the module, provided you tell it to do so by answering 'y' to the prompt. *If you hurry at this point, you can lose data.*

At the end of your session, you can

```
work/test>  cvs logout
```

which will remove the entry for the given CVS server from the file '*~/.cvspass*'. This means that the next time you want to access the server again from the same machine, you will be asked for the CVS password again.

6 Tags and revision numbers

CVS identifies revisions with unique version numbers, like 1.1.1.4 or 2.15. Often it is much more convenient to refer to a given revision with a symbolic tag. You can attach a tag to your working copy with

```
work/test>  cvs rtag jets-hydro-5 test
```

Do not use rtag in the form 'cvs rtag hydro-5.' — rtag needs the module name as last argument and will otherwise tag all the files you have under CVS control, affecting any other modules as well.

There is also a command 'cvs tag'. The bottom line is that you use *tag* to tag individual files, but *rtag* for the whole module.

> *A more detailed discussion of the differences between tag and rtag is given in one of the FAQs (http://www.loria.fr/~molli/fom-serve/cache/211.html): The end result of both commands is that a [tag], or symbolic name, is attached to a single revision in each of a collection of files. The differences lie in:*
>
> - *The collection of files they work on.*
>
> *"rtag" works on the collection of files referred to by a "module" name as defined in the "modules" file, or a relative path within the Repository.*
>
> *"tag" works on files and directories specified on the command line within the user's working directory. (Default is '.')*

Both commands recursively follow directory hierarchies within the named files and directories.

– *The revisions they choose to tag.*

"rtag" places a tag on the latest committed revision of each file on the branch specified by the '-r' option. By default it tags the Main Branch.

"tag" places a tag on the BASE (i.e. last checked out, updated or committed) revision of each file found in the working directory. (The BASE revision of a file is the one stored in the ./CVS/Entries file.)

[...]

If you want to bring all your files up to revision 3.0 (including those that haven't changed), you might invoke

```
work/test>  cvs commit -r 3.0
```

This only works if none of the files in the module had a revision number higher than 3.0. It is probably a good idea to check with your collaborators before you decide to increase the major revision number (the first digit of the revision number).

7 Conflicts

If you want to *commit* a file (say, *run.csh*), but someone else has in the meantime committed a later version of it than the one you were working with, '*commit*' will speak very roughly to you:

```
unix>  cvs commit -m 'Removed a few module references'
cvs commit: Examining src
cvs commit: Up-to-date check failed for `src/run.f90'
cvs [commit aborted]: correct above errors first!
```

What you should do now is *update* the file '*src/run.f90*' (or the whole directory '*src*'):

```
unix>  cvs update src
cvs server: Updating src
RCS file: /home/cvs/cvsroot/test/src/run.f90,v
retrieving revision 2.0
retrieving revision 2.1
Merging differences between 2.0 and 2.1 into run.f90
M src/run.f90
```

If you are lucky — i.e. if the changes appeared in different files, or even if they are located in non-overlapping regions of the same file — the two versions are automatically merged and everything is OK. If you have doubts, take a look a the merged file. (If you are unlucky, you must manually resolve the conflict, see below.) Now *src/run.f90* contains both modifications together[1], and you can commit the merged file:

[1] If this is not what you wanted, you can reconstruct your version of *test/src/makefile* with '`cvs update -j ...`'

```
unix>  cvs commit -m "Made important changes and merged" src
Checking in run.f90;
/var/local/cvsroot/test/src/run.f90,v  <--  run.f90
new revision: 1.3; previous revision: 1.2
done
```

If you are *really* lucky, the merged code still compiles...

However, if you and your colleague have modified the same part of the code, the conflict can not be resolved automatically by merging, and you obtain a warning

```
unix>  cvs update
cvs server: Updating src
RCS file: /home/cvs/cvsroot/test/src/run.f90,v
retrieving revision 2.1
retrieving revision 2.2
Merging differences between 2.1 and 2.2 into run.f90
rcsmerge: warning: conflicts during merge
cvs server: conflicts found in src/run.f90
C src/run.f90
```

Now you have to resolve the conflict manually by editing the file, fixing it and running '*cvs commit*.' The file *src/run.f90* looks like this:

```
...
!
    use Mpicomm
<<<<<<< run.f90
!    use Cdata
!    use Deriv
=======
    use Cdata ! Really use it
    use Deriv
>>>>>>> 2.2
    use Sub
    use Timestep
...
```

The line between '`<<<<<<<`' and '`=======`' represents your changes of *run.csh*, while the part between '`=======`' and '`>>>>>>>`' has been committed in version 2.2 of the file by your swift colleague.

You can find your version of the file in the hidden file '`src/.#run.csh.2.1`'.

8 Flags issued by '`update`'

In the previous example, '*update*' flagged *test/src/makefile* with the capital letter '`M`' and *test/run.csh* with '`C`'. Here is a list of all the flags used by '*update*'.

Flag	Meaning
U	File was updated from the repository.
P	Essentially the same as 'U' (but the server sends a patch, rather than the whole file)
A	File has been added to your private copy of the sources. This is a reminder that the file needs to be *commit*ted.
R	File has been removed from your private copy of the sources. This is a reminder that the file needs to be *commit*ted.
M	File is modified in your working directory. It had either not been modified in the repository, or your changes and those in the repository have been successfully merged.
C	A conflict occurred. An unmodified copy of your file is saved as '`.#file.version`' in your working directory where *version* is the revision that your modified file started from.
?	File is in your working directory, but not in the repository, nor is it in the list of files for CVS to ignore. You probably want to *add* it.

9 CVS/RCS Keywords

When you check in files, CVS automatically expands strings of the form '$Author$', '$Date$', 'Id', etc. with information about the file. In particular, 'Id' will be expanded to something like

```
$Id: cvs.tex,v 1.25 2005/11/28 22:32:29 dobler Exp $
```

which is often quite useful to have somewhere in your text files. You can even print this string in your code or include it in a LaTeX file.

Using CVS Keyword Expansions in LaTeX [after `http://atom.ecn.purdue.edu/~notz/latex-cvs.html`] Using CVS keywords in a LaTeX document is not straightforward, since the dollar sign switches to mathematical mode if no measures are taken. There are two common workarounds and two LaTeX-packages:

1. Encapsulate the keyword line by a `\verb$$` environment:

   ```
   \verb$Id: cvs.tex,v 1.25 2005/11/28 22:32:29 dobler Exp $
   ```

   or

   ```
   \verb|$Id: cvs.tex,v 1.25 2005/11/28 22:32:29 dobler Exp $|
   ```

   (depending on whether you want the dollar signs to be printed or not). This approach is subject to the limitations of the `\verb` environment, which, e. g. cannot be an argument to a LaTeX macro.

2. Put '$␣' and '␣$' around the keyword line:

   ```
   $ $Id: cvs.tex,v 1.25 2005/11/28 22:32:29 dobler Exp $ $
   ```

   This neutralises the Dollar signs (by creating two math environments containing only one blank) and allows the following text to be used as you like — including the font of your choice and handing it over to a macro.

3. The two packages *rcs* and *rcsinfo* from *CTAN* allow the inclusion of RCS keywords in LaTeX documents

10 Creating a repository

If you want to create a new repository, you use *cvs init*:

```
unix>  cvs -d ~/cvsroot init
```

— this creates a repository in your home directory. To access this repository, you should set CVSROOT accordingly:

```
unix>  setenv CVSROOT ~/tmp/cvsroot
```

11 Nota Bene

- If you are uncertain about what a given command might do, run it with '`cvs -n <command>`' first.

  ```
  unix>  cvs -n update  (Does not change any file)
  unix>  cvs -n commit  (Does not change any file)
  ```

  The '`-n`' flag tells CVS to do a 'dry run' of the command and not change any files.

- When creating new directories, remember to explicitly *add* them. The commands '*update*' and '*release*' will show the new directory flagged with a question mark, but nobody will keep you from finally deleting your working copy and thus getting rid of all the files created in your new directory.

- If you rename files, you must *remove* the old file and *add* the new one:

  ```
  unix>  mv old new
  unix>  cvs add new
  unix>  cvs remove old
  ```

  Remember that for these changes to take place in the repository, you must still *commit* them.

  > *The new file will know nothing about the modification history prior to this operation. If you want to rename a file, retaining the full history, then you need direct access to the repository:*[2] *You* copy *the file from 'old,v' to 'new,v'. Then you do* cvs remove *on the old version. This ensures that* cvs update *removes the old version.*

- Before you *release* a modified working copy, you must *commit* it — otherwise, you get warnings about modified files (marked with '`M`' in front of the file name) and should then definitely not continue the *release*, unless you want to lose the changes you have made.

- Keep in mind that CVS simply ignores symbolic links. However, there should be no need to link the *src/* directory and files any more, since all the supposed advantages of this technique are features of CVS.

- Do not forget the leading `cvs` for the CVS commands. Otherwise you might end up with cryptic error messages like in the following example

[2] This trick is due to Karl Fogel's book mentioned in Section 3.

```
unix>  co start.csh
co: RCS/start.csh,v: No such file or directory
```

This is an error message from RCS (another version control system), the '`co`' command of which you called by accident. Since RCS and CVS are somehow related (although CVS seems to be no longer built on top of RCS) and CVS indeed works with files like '*start.csh,v*', you might be tempted to take the error message for meaningful.

– Do not edit lines of the form

```
$Id: cvs.tex,v 1.25 2005/11/28 22:32:29 dobler Exp $
$Author: dobler $
$Date: 2005/11/28 22:32:29 $
$Revision: 1.25 $
```

As discussed in Section 9, they are automatically updated by CVS each time you commit or update the corresponding files.

– The revision number of your module does in general not coincide with those of the files therein. As an example, some of the source files that make up RCS 5.6 have the following revision numbers:

ci.c	5.21
co.c	5.9
ident.c	5.3

– You can specify dates (with the '`-D`' option) in a variety of formats.

These two types of format are preferred:

```
unix>  cvs co -D '22 Aug 2001'
unix>  cvs co -D '22 Aug 2001 20:05'
unix>  cvs co -D '2001-08-22'
unix>  cvs co -D '2001-08-22 20:05'
```

However, the following work as well:

```
unix>  cvs co -D 'August 22 2001 20:05pm'
unix>  cvs co -D 'a fortnight ago'
unix>  cvs co -D 'yesterday'
unix>  cvs co -D '1 hour ago'
```

So if you want to see what you have done during the previous hour, type

```
unix>  cvs diff -D '1 hour ago' src/run.f90
```

– It makes sense to always *update* immediately before you *commit* any changes. It is not terrible, though, if you don't. You might just get warnings (and disobedience) from '*commit*'.

– Only commit versions that compile and run. The socially acceptable minimum is to commit a version that at least compiles successfully.

If someone else has made changes simultaneously and your *update*d code doesn't compile any more, either fix this problem before *commit*ting, or create a separate branch for your version of the code.

– Be minimalistic about the files you keep in the working directory. Remember that there is no need to retain files (say, IDL programs) that were once useful and might, perhaps, possibly, under special circumstances be needed again in the distant future.

  In particular, do not keep old versions of files in your working directory. For example, if you modify version 1.5 of '*run.f90*' in an experimental way, just change it; if you then need the original version, retrieve it with '`cvs co -r 1.5 test run.f90`'

– To put a new project under CVS control, go to its top directory and `import` it:

```
unix>  cd ~/f90/projects/solitons
solitons>  cvs import -m "Import of soliton code v. 0.05" \
                      f90/solitons ncl-mhd Solitons_0-05
```

  Here '*f90/solitons*' is the name of the module in the repository, '*ncl-mhd*' is a "vendor tag" (unimportant in our case) and '*Solitons_0-05*'[3] is a *tag* attached to this imported revision, allowing to refer to it later.

  The files are imported with revision number 1.1.1.1, which is specific for the vendor branch. At the same time, the files have revision number 1.1 and only this is used for the version on the trunk. So, the first time you modify an imported file, its revision number gets increased to 1.2, the next time to 1.3, and so on.

  Importing a directory does not make it a checked out version of it (i. e. the directory where you called *cvs import* will not contain a '*CVS*' subdirectory. One simple way of turning your '*solitons*' directory into a CVS-controlled one after the import is to do

```
projects/solitons>  cd ..
projects>  mv solitons solitons-deleteme-eventually
projects>  cvs co -d solitons f90/solitons
```

  You can keep the original directory around for some time in case you forgot to check in some files, but from now on you will work with the checked-out version.

12 My top ten CVS commands

Here are my top ten commands, i. e. (ordered by frequency of use) the main CVS commands in my repertoire:

1. `cvs -qn update`
2. `cvs update -d`

[3] '*Solitons-0.05*' would look much nicer (in my opinion), but tags must not contain any of the characters '`$,.:;`'

3. `cvs commit`
4. `cvs diff <file>` *(difference to original version)*
   or
   `cvs diff -r HEAD <file>` *(difference to latest repository version).*

   A variant is
   `cvs diff -u -rHEAD <file> | a2ps -Eudiff --prologue=diff -Pdisplay`
   to pretty-print difference between local and latest version
5. `cvs checkout <module>`
6. `cvs add <file>`
7. `cvs checkout -d <directory> <module>`
   (check out module into specific directory)
8. `cvs log <file> | less`
9. `cvs annotate <file>`
10. `cvs import <repository> <vendor-tag> <release-tag>`

Note: The `-q` flag ('be somehow quiet') is so useful for larger projects that you may want to put the line

```
cvs -q
```

into '*~/.cvsrc*' to have it always set.

13 Other user interfaces

1. **VC (minor) mode:** If Emacs is your operating system of choice, you can use *VC* mode as a front end to *CVS*. Normally, Emacs (at least versions ≥ 21) automatically detects which files are under *CVS* and adds a string like "CVS:1.15" to your mode line.

   Useful key strokes are

   `C-x C-q` and **`C-x v v`:** *vc-next-action*, do *cvs commit* or *cvs update*, whichever makes more sense

   `C-x v i`: *vc-register*, i. e. *cvs add*

   `C-x v =`: *vc-diff*, does *cvs diff* on buffer file

   `C-x v l`: *vc-print-log*, shows output from *cvs log* in separate buffer

   Less essential, but useful key strokes are

   `C-x v u`: *vc-revert-buffer*, reverts to the version buffer file was based on (i. e. undoes all changes)

C-x v ~: *vc-version-other-window*, loads a specific version into another buffer (allows for *ediff*)

C-x v a: *vc-update-change-log*, extracts log information and writes or adds it to a ChangeLog file

C-x v h: *vc-insert-headers*, inserts 'Id' as a comment

C-x v g: *vc-annotate*, do *cvs annotate* with colors indicating different versions...

2. **pcl-cvs:** There is another *CVS* front end for Emacs, called *pcl-cvs*. I do not use it and think that *VC* mode is the way to go, but if you are interested in *pcl-cvs*, here is a short description.

   To get started, just type '`M-x cvs-update RET`' and enter the name of a directory where you have a checked-out CVS module:

```
PCL-CVS release 1.05 from CVS release $Name:  $.
Copyright (C) 1992, 1993 Per Cederqvist
Pcl-cvs comes with absolutely no warranty; for details consult the manual.
This is free software, and you are welcome to redistribute it under certain
conditions; again, consult the TeXinfo manual for details.

In directory /home/dobler/f90/mhdf/work/test:
  Updated     run.csh

In directory /home/dobler/f90/mhdf/work/test/src:
  Modified ci run.f90
  Updated     mhd1.f90
  Unknown     mhd2.f90
---------- End -----
```

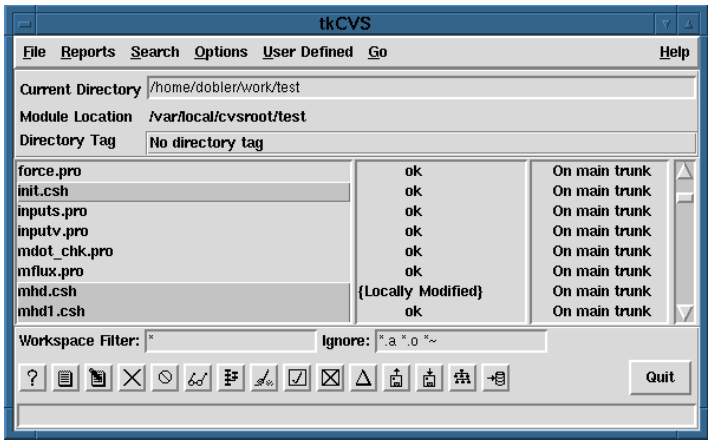
   Now you can do lots of fancy things with a few key strokes. Look at the *info* documentation for *pcl-cvs* for details.

   One of the good points about *pcl-cvs* is that you have the marvelous tools '*ediff*' and '*emerge*' at hand (and even automatically invoked) if you need them.

   On my *Debian* system, I had problems getting pcl-cvs to run. See `http://www.kis.uni-freiburg.de/~dobler/docs` for how I solved them.

   Warning to Vi users: There is a reported case of a Vi user who converted to Emacs just to be able to use *pcl-cvs*; so better watch out.

3. **tkcvs:** If you prefer graphical user interfaces (the ones where you have to mechanically repeat the same sequence of 20 mouse clicks all of the time), try *tkcvs*.



You will find it via the *cvshome* web page; it should be possible to install it on any Unix machine running a recent version of *tcl/tk*.

4. There are also graphical clients for Macintosh (*MacCVSClient*, see `http://www.cvshome.org/cyclic/cvs/mac.html`) and Windows (*WinCVS*, see `http://www.cvshome.org/cyclic/cvs/windows.html`)

Appendix

A Overview over CVS commands

The following overview over the basic CVS commands has been adapted from the 'CVS tutorial' (`http://www.loria.fr/~molli/cvs/cvs-tut/cvs_tutorial_toc.html`) by Gray Whatson.

Most of the below commands should be executing while in the directory you checked out. If you did a '`cvs checkout malloc`' then you should be in the malloc sub-directory to execute most of these commands. '`cvs release`' is different and must be executed from the directory above.

`cvs add` and **`cvs remove`**
It can be that the changes you want to make involve a completely new file, or removing an existing one. The commands to use here are:

`cvs add` *'filename'*
`cvs remove` *'filename'*

You still have to do a '*commit*' after these commands to make the additions and removes actually take affect. You may make any number of new files in your copy of the repository, but they will not be committed to the central copy unless you do a '`cvs add`'.

CVS remove does not actually remove the files from the repository. It only removes them from the "current list" and puts the files in the CVS Attic. When another person checks out the module in the future they will not get the files that were removed. But if you ask for older versions that had the file before it was removed, the file will be checked out of the Attic.

`cvs admin`
This is the CVS interface to assorted administrative facilities. Some of them have questionable usefulness for CVS but exist for historical purposes. Some of the questionable options are likely to disappear in the future. This command *does* work recursively, so extreme care should be used.

`cvs annotate`
Gives you an annotated listing of the current version of a file, containing for each line information about in which version, by whom and when it was written. It does *not* contain information about deleted or modified lines (to get this, use '`cvs diff`' on the two versions you are interested in).

```
unix>  cvs annotate start.csh
Annotations for start.csh
***************
1.1 (brandenb 22-Apr-99): ! src/start.x
1.2 (nbmvr    10-Jul-99):
```

```
1.2 (nbmvr    10-Jul-99): -8.,8.,              :zmin,zmax
1.2 (nbmvr    12-Jul-99): .05,2.,1.,40060000., :rin,rqu,xboxmax,rLL
1.1 (brandenb 22-Apr-99): -.25,1.5,.1499,8,    :r1,r2,height,nwidth
1.1 (brandenb 22-Apr-99): 1.,0,0,0,            :mu0,B0,Bphi0,eps_quadru
1.1 (brandenb 22-Apr-99): 1.666667,.1,0.,      :gamma,beta,HH0
1.1 (brandenb 22-Apr-99): 0,0,                 :nsmooth,nsmoothrun
1.1 (brandenb 22-Apr-99): 1.,0.1,              :frac1,dlmask
1.3 (dobler   12-Jul-99): 1,2.,0               :isymm,scale,iffree
1.1 (brandenb 22-Apr-99): 0.0000001,           :ampl
1.1 (brandenb 22-Apr-99): EOF
1.1 (brandenb 22-Apr-99):
1.1 (brandenb 22-Apr-99): rm -f tmp/n.dat
1.1 (brandenb 22-Apr-99): rm -f t*.dat
1.1 (brandenb 22-Apr-99):
1.1 (brandenb 22-Apr-99): #
1.3 (dobler   12-Jul-99): #  iffree  --  initialise B force-free
```

cvs checkout (or cvs co)
To make a local copy of a module's files from the repository execute '`cvs checkout module`' where module is an entry in your modules file (see below). This will create a sub-directory module and check-out the files from the repository into the sub-directory for you to work on.

cvs commit
When you think your files are ready to be merged back into the repository for the rest of your developers to see, execute '`cvs commit`'. You will be put in an editor to make a message that describes the changes that you have made (for future reference). Your changes will then be added to the central copy.

When you do a '*commit*', if you haven't updated to the most recent version of the files, CVS tells you this; then you have to first *update*, resolve any possible clashes, and then redo the *commit*.

cvs diff
To see the differences between your version of the files, and the version in the repository you started from, do:

`cvs diff` *'filename(s)'*

If you want to compare to the latest version in the repository, use

`cvs diff -r HEAD` *'filename(s)'*

cvs history
To find out information about your CVS repositories use the '`cvs history`' command. By default '*history*' will show you all the entries that correspond to you. Use the '`-a`' option to show information about everyone.

cvs history -a -o
shows you (a)ll the checked (o)ut modules

cvs history -a -T

reports (a)ll the r(T)ags for the modules

cvs history -a -e
reports (a)ll the information about (e)verything

cvs import
Use 'import' to incorporate an entire source distribution from an outside source (e.g., a source vendor) into your source repository directory. You can use this command both for initial creation of a repository, and for wholesale updates to the module from the outside source. *Note Tracking sources::, for a discussion on this subject

cvs init
Create a CVS repository if it doesn't exist.

cvs log
To see the commit messages for files, and who made them, use:

`cvs log` *'filename(s)'*

cvs login, logout
Connect to, and disconnect from, the CVS server when using the *server/client* mode of accessing the repository (which we do).

cvs rdiff
Create 'patch' format diffs between releases

Builds a Larry Wall format patch(1) file between two releases, that can be fed directly into the 'patch' program to bring an old release up-to-date with the new release. (This is one of the few CVS commands that operates directly from the repository, and doesn't require a prior checkout.) The diff output is sent to the standard output device.

cvs release
When you are done with your local copy of the files for the time being and want to remove your local copy use '`cvs release module`'. This must be done in the directory above the module sub-directory you which to release. It safely cancels the effects of '`cvs checkout`'. Usually you should do a commit first.

If you wish to have CVS also remove the module sub-directory and your local copy of the files then you do '`cvs release -d module`'.

NOTE: Take your time here. CVS will inform you of files that may have changed or it does not know about (watch for the '?' lines) and then will ask you to confirm this action. Make sure you want to do this.

cvs remove
See *cvs add*.

cvs rtag
Like '*tag*', '*rtag*' marks the current versions of files but it does not work on your local copies but on the files in the repository. To tag all my libraries with a version name I can do:

```
cvs rtag LIBRARY_2_0 lib
```

This will recursively go through all the repository directories below lib and add the `LIBRARY_2_0` tag to each file. This is one of the most useful features of CVS. Use this feature if you are about to release a copy of the files to the outside world or just want to mark a point in the developmental progression of the files.

cvs status
Show current status of files: latest version, version in working directory, whether working version has been edited and, optionally, symbolic tags in the RCS file. (Does not change repository or working directory.)

cvs tag
One of the exciting features of CVS is its ability to mark all the files in a module at once with a symbolic name. You can say 'this copy of my files is version 3'. And then later say 'this file I am working on looked better in version 3 so check out the copy that I marked as version 3.'

Use `cvs tag` to tag the version of the files that you have checked out. You can then at a later date retrieve this version of the files with the tag.

```
cvs tag tag-name filenames
```

Later you can do:

```
cvs co -r tag-name module
```

cvs update
To update your copy of a module with any changes from the central repository, execute '`cvs update`'. This will tell you which files have been updated (their names are displayed with a '`U`' before them), and which have been modified by you and not yet committed (preceded by an '`M`').

It can be that when you do an update, the changes in the central copy clash with changes you have made in your own copy. You will be warned of any files that contain clashes by a preceding '`C`'. Inside the files the clashes will be marked in the file surrounded by lines of the form `<<<<<<<` and `>>>>>>>`. You have to resolve the clashes in your copy by hand. After an update where there have been clashes, your original version of the file is saved as '`.#file.version`'.

If you feel you have messed up a file and wish to have CVS forget about your changes and go back to the version from the repository, delete the file and do an '`cvs update`'. CVS will announce that the file has been "lost" and will give you a fresh copy.

With option '`-d`', create any directories that exist in the repository if they're missing from the working directory. Normally, 'update' acts only on directories and files that were already enrolled in your working directory.

cvs edit,editors,watch,watchers,unedit
These are commands that are irrelevant for us.

B Branches

B.1 Accessing branches

CVS allows different branches of one module to be worked on simultaneously. You can branch from an earlier version, work on that branch and finally merge your changes into the latest revision on the main branch.

To check out the branch labelled '*S-const-branch*' of module '*test*', type

```
work/test>  cvs update -r S-const-branch
```

(or '`cvs co -r S-const-branch`' if you do not have a working copy).

If you now commit changes, they will be saved on the branch '*S-const-branch*':

```
work/test>  cvs commit
[ ... ]
work/test>  cvs status
===================================================================
File: start.csh          Status: Up-to-date

   Working revision:    1.32.2.3
   Repository revision: 1.32.2.3     /var/local/cvsroot/test/start.csh,v
   Sticky Tag:          S-const-branch (branch: 1.32.2)
[ ... ]
```

The version number 1.32.2 is the number of the branch that was split off revision 1.32. Note that the branch tags stick to the branch (i. e. checking out the version with the tag '*S-const-branch*' will always give you the latest version on that branch), while revision tags are tied to one revision (like e. g. 1.32), although you can update them if you like.

'`cvs log`' lists you the tags, including branch tags:

```
work/test>  cvs log start.csh
RCS file: /var/local/cvsroot/test/start.csh,v
Working file: start.csh
head: 1.35
branch:
locks: strict
access list:
symbolic names:
        S-const-branch: 1.32.0.2
        pre-S-const-branch: 1.32
keyword substitution: kv
total revisions: 38;    selected revisions: 38
```

```
description:
----------------------------
revision 1.35
[ ... ]
----------------------------
revision 1.32
branches:  1.32.2;
[ ... ]
```

B.2 Creating branches

[from the FAQ]:

Suggested technique:

1. Attach a non-branch tag to all the revisions you want to branch from (i.e. the branch point revisions).
2. When you decide you really need a branch, attach a branch tag to the same revisions marked by the non-branch tag.
3. "Checkout" or "update" your working directory onto the branch.

Schematically, this means

(Write information about tags-to-come to Tags.list and commit)

```
unix>  cvs rtag <branch_point_tag> <module>
unix>  cvs rtag -b -r <branch_point_tag> <branch_tag> <module>
unix>  cvs checkout -r <branch_tag> <module>
```

The first step refers to the case where you are keeping a list of tags in a file '*Tags.list*'; you should update this file before you branch, so the information about the branch points is up to date on both trunk and branch.

C Tips, tricks and troubleshooting

C.1 User level tips and tricks

How can I *checkout* a directory without getting all its subdirectories? Use the '`-l`' flag of *checkout* (or *update*) to avoid recursion through the directory tree:

```
unix>  cvs co -l -d runs pencil-runs
```

To only get a sparse tree, say `runs/forced/halo1/`, you will have to apply this technique sequentially:

```
unix>  cvs co -l -d runs pencil-runs
unix>  cd runs;   cvs up -dl forced
unix>  cd forced; cvs up -dl halo1
```

The combination of '`-l`' and '`-d`' creates subdirectories without recursing.

My cvsroot has changed (new server name, ...) – how do I update my checked out copies?

For each copy, *cd* to the top directory, then do

```
unix>  oldroot=':pserver:USER\@OLD.HOST.DOM:/OLD/PATH'
unix>  newroot=':pserver:USER\@NEW.HOST.DOM:/NEW/PATH'
unix>  find . -path '*CVS/Root' \
        | xargs fgrep -l "${oldroot/\\\\@/@}" \
        | xargs perl -i.bak -pe "s{$oldroot}{$newroot}"
```

where (you guessed it) you replace all uppercase names and paths with real stuff. If you run this once, it creates a backup '*Root.bak*' of each '*Root*' file it adapts. When running a second time, however, the first backup will get overwritten.

C.2 Administration

C.2.1 Problems with the CVS pserver

Here is a checklist that proved useful.

1. Have you set up your repository correctly?

   `cvs -d /home/User/CVS init` *(or wherever the repository should go)*

   You *do* need the '-d ...', since otherwise CVS takes the value from CVSROOT — which points to a directory that is not yet set up for CVS.

If you try to `cvs login`, but get no connection:
`cvs [login aborted]: connect to ...:2401 failed: Connection refused`

2. (From the Cederqvist manual)
   Try

   `telnet servername 2401`

   After connecting, send any text (for example "foo" followed by return). If CVS is working correctly, it will respond with

   `cvs [pserver aborted]: bad auth protocol start: foo`

3. Does your system know about the *service* cvs? If '*/etc/inetd.conf*' operates with service names instead of port numbers (i. e. if the first entry of each inetd line is a name, rather than a number) your cvs entry there,

   ```
   cvspserver stream tcp nowait root \
   /usr/local/bin/cvs cvs --allow-root=/home/User/CVS pserver
   ```

   — then '*/etc/services*' must define the service *cvspserver*:

   ```
   cvspserver      2401/tcp
   ```

   to tell inetd to start cvs when there is a request on port 2401.

4. Have you restarted *inetd* after changing '*/etc/{inetd.conf,services}*'?

```
Linux>  /usr/bin/killall -HUP inetd
IRIX>  /sbin/killall -HUP inetd
```

5. Verify your tcp wrapper settings (see '`man hosts_access`', '`man hosts_options`' under Linux):

   ```
   Linux>  /usr/sbin/tcpdmatch cvs localhost
   IRIX>  /usr/etc/tcpdmatch cvs localhost
   ```

   If access is 'granted', this part of the setup is OK. If access is 'denied', set up your '*/etc/hosts.{allow,deny}*' correctly

If you try to `cvs login`, but get an authorisation error:

```
cvs [login aborted]: authorization failed: server ... rejected access
```

6. Have you set up a password file '*passwd*' in '*/home/User/CVS/CVSROOT*'?
7. Is the CVS repository correctly specified in both, '*/etc/inetd.conf*' and your environment variable? The tilde does not work here, thus

   ```
   cvspserver ... cvs --allow-root=~User/CVS
   ```

   must be replaced by

   ```
   cvspserver ... cvs --allow-root=/home/User/CVS
   ```

   if *User*'s home directory is '*/home/User*'.

   Similarly, in your ~/*.cshrc* file, you should use

   ```
   setenv CVSROOT :pserver:$USER@server.domain:/home/User/CVS
   ```

8. Check the system log files ('*/var/adm/SYSLOG*' under *IRIX*; '*/var/log/{message,syslog,}*' under Linux) for why *inetd* rejected the access

Weirder problems

9. You receive a complaint about an unrecognised option:

   ```
   cvs [login aborted]: unrecognized auth response from ...: \
     cvs: unrecognized option `--allow-root=...'
   ```

   Are you running version 1.9 or older of *CVS*? In that case, *cvs* does not understand the `--allow-root` option. Just drop it.

— Written July 2, 2006 by Wolfgang Dobler <Wolfgang.Dobler@ucalgary.ca> —

G.1 Appendix

G.1.1 Lab exercises

Notes:

- Set the `CVSROOT` environment variable to `:pserver:`*USER*`@obelix.capca.ucalgary.ca:/repos/phys535`
- You may want to set the environment variable `CVSEDITOR` or `EDITOR` to make sure CVS won't start an unknown editor for you.

Question 51 *Playing with CVS I*

(a) Check out the `test` repository from the CVS server.

(b) Modify a file or two, add a new file and delete one. Commit your changes.

(c) Edit the file `magnetic.f90` (do not yet commit your changes):

- Rename the variable `iaa` to `ieee` — student A
- Exchange the order of the routines `initialize_magnetic` and `init_aa` — student B
- Replace all single quotes (') by double quotes (") — student C
- Replace calls to the `beltrami()` routine by calls to `tortellini()` — student D

(d) When everybody has finished their changes, commit them.

(e) What were the changes between revisions 1.1 and 1.3 of '`scripts/mkcparam`'?

Question 52 *Playing with CVS II*

(a) Create a toy directory `mytest` and populate it with a few files. Import it under `USERNAME/mytest`. Look at the directory structure in the repository.

(b) Now turn your local copy of `mytest` into a directory under CVS control.

(c) Add two file, `toto` and `toto.bak` and do '`cvs update`'. Can you guess why CVS would behave like this?

(d) Add useful entries to `~/.cvsignore` (look at mine for inspiration) and verify that they work.

Question 53 *Making use of CVS*

(a) Import your IDL files into your CVS repository.

Before you start. think about a useful directory structure (renaming and moving files should be kept to a minimum with CVS).

(b) Add useful keywords to some of your files and '`cvs commit`'. Verify the result.