CVS - Concurrent Versions System
Harvard CfA, Ay 192, Spring 2008
Doug Finkbeiner

Version Control

Version control software allows software and documents to be modified by different users on different computers without stepping on each other's toes. It allows users to:

- share and collaborate on LaTeX documents and code,
- simultaneously modify different parts of the same software package,
- retain all previous revisions of one's files, and
- keep files on different machines (e.g. laptop and desktop) synchronized.

CVS is one of the available version control packages. The source code, executables, and documentation may be downloaded from here.[1] There are other forms of version control software you might want to look into (Subversion[2], Mercurial[3], and git[4] are leading contenders) but we will not talk about them further in this document. The next section is a paraphrase of material from the CVS web site - thanks to Roland Pesch, David Grubbs, and others.

What is CVS?

CVS is a version control system. Using it, you can record the history of your source files.

For example, bugs sometimes creep in when software is modified, and you might not detect the bug until a long time after you make the modification. With CVS, you can easily retrieve old versions to see exactly which change caused the bug. This can sometimes be a big help.

You could of course save every version of every file you have ever created, but this is confusing (thesis.tex.27sep anyone?). CVS stores all the versions of a file in a single file in a compact way that only stores the differences between versions.

CVS also helps you if you are part of a group of people working on the same project. It is all too easy to overwrite each others' changes unless you are extremely careful. Some editors, like GNU Emacs, try to make sure that the same file is never modified by two people at the same time. Unfortunately, if someone is using another editor, that safeguard will not work. CVS solves this problem by insulating the different developers from each other. Every developer works in his own directory, and CVS merges the work when each developer is done.

[1] http://www.cvshome.org
[2] http://subversion.tigris.org/
[3] http://www.selenic.com/mercurial/wiki/
[4] http://git.or.cz/

What CVS is NOT

CVS can do a lot of things for you, but it does not try to do *everything.*

CVS is not a build system. We are using CVS for IDL .pro files which are never "built" with a "Makefile" like C and FORTRAN programs are. However, compiled programs are often mixed with IDL in practice, so keep the following in mind. Though the structure of your repository and modules file interact with your build system (e.g. 'Makefile's), they are essentially independent. CVS does not dictate how you build anything. It merely stores files for retrieval in a tree structure you devise. CVS does not dictate how to use disk space in the checked out working directories. If you write your 'Makefile's or scripts in every directory so they have to know the relative positions of everything else, you wind up requiring the entire repository to be checked out. If you modularize your work, and construct a build system that will share files (via links, mounts, VPATH in 'Makefile's, etc.), you can arrange your disk usage however you like. But you have to remember that any such system is a lot of work to construct and maintain. CVS does not address the issues involved. Of course, you should place the tools created to support such a build system (scripts, 'Makefile's, etc) under CVS. Figuring out what files need to be rebuilt when something changes is, again, something to be handled outside the scope of CVS. One traditional approach is to use make for building, and use some automated tool for generating the dependencies which make uses. See section How your build system interacts with CVS (at cvshome.org), for more information on doing builds in conjunction with CVS.

CVS is not a substitute for management. In a large project, your collaborators, and project leaders are expected to talk to you frequently enough to make certain you are aware of schedules, merge points, branch names and release dates. If they don't, CVS can't help. CVS is an instrument for making sources dance to your tune. But you are the piper and the composer. No instrument plays itself or writes its own music.

CVS is not a substitute for developer communication. When faced with conflicts within a single file, most developers manage to resolve them without too much effort. But a more general definition of "conflict" includes problems too difficult to solve without communication between developers. CVS cannot determine when simultaneous changes within a single file, or across a whole collection of files, will logically conflict with one another. Its concept of a conflict is purely textual, arising when two changes to the same base file are near enough to spook the merge (i.e. diff3) command. CVS does not claim to help at all in figuring out non-textual or distributed conflicts in program logic. For example: Say you change the arguments to function X defined in file 'A'. At the same time, someone edits file 'B', adding new calls to function X using the old arguments. You are outside the realm of CVS's competence. This type of problem is only solved by reading specs and talking to your collaborators.

CVS Basics

In CVS there is a concept of checking out a package or "module" (set of files in a directory or set of directories) from a "repository" and then checking in (or "committing") changes. Each developer has a separate copy of the code to edit, and keeps this in sync with the repository with the "commit" and "update" commands.

Sample Session

For now we will assume a repository is set up and environment variables (see below) are set properly. A typical CVS session might go something like this:

```
> cvs checkout idlutils  - check out a module called idlutils

> cd idlutils
> emacs flipper.pro   - edit existing file
...
> cvs update          - make sure nobody else has changed flipper.pro
> cvs commit flipper.pro
Now the editor named in $CVSEDITOR is invoked to allow you to type
a comment like
‘‘Fixed bug introduced by Alex in v. 1.12’’
Alternatively, one may type
> cvs commit -m ‘Fixed bug introduced by Alex in v. 1.12’ flipper.pro
to avoid launching the editor.  The file can also be committed from
within EMACS (see below).

> emacs highz.pro  - create a new file
> cvs add highz.pro
> cvs commit -m 'new subroutine of flipper.pro' highz.pro
...
Suppose now you edit many files in the current directory to implement
a new feature.  You may commit ALL of the changes with
> cvs update           - make sure nothing else changed
This will warn you that you have changed files and they are now out of
synch with the repository.

> cvs commit -m 'added new feature' *
This commits all changed files in the directory.
Note: cvs will ignore requrests to commit files whose contents are
unchanged since the last update.

To see the line-by-line history of a file, you can type
> cvs annotate atv.pro    (example from ATV in idlutils)

1.9          (schlegel 14-Apr-00): case eventchar of
1.9          (schlegel 14-Apr-00):     '1': atv_move_cursor, eventchar
1.9          (schlegel 14-Apr-00):     '2': atv_move_cursor, eventchar
1.9          (schlegel 14-Apr-00):     '3': atv_move_cursor, eventchar
```

```
1.9          (schlegel 14-Apr-00):     '4': atv_move_cursor, eventchar
1.9          (schlegel 14-Apr-00):     '6': atv_move_cursor, eventchar
1.9          (schlegel 14-Apr-00):     '7': atv_move_cursor, eventchar
1.9          (schlegel 14-Apr-00):     '8': atv_move_cursor, eventchar
1.9          (schlegel 14-Apr-00):     '9': atv_move_cursor, eventchar
1.9          (schlegel 14-Apr-00):     'r': atv_rowplot
1.17         (dfink    24-Jun-03):     'R': atv_rowplot, /overplot
1.9          (schlegel 14-Apr-00):     'c': atv_colplot
1.17         (dfink    24-Jun-03):     'C': atv_colplot, /overplot
1.9          (schlegel 14-Apr-00):     's': atv_surfplot
1.25         (blanton  04-Aug-06):     'm': atv_markpoint
1.26         (blanton  08-Aug-06):     'd': atv_displaypoint
1.25         (blanton  04-Aug-06):     'k': atv_killpoint
1.9          (schlegel 14-Apr-00):     't': atv_contourplot
1.11         (schlegel 01-Dec-00):     'p': atv_apphot
1.9          (schlegel 14-Apr-00):     'i': atv_showstats
1.11         (schlegel 01-Dec-00):     'q': atv_shutdown
1.9          (schlegel 14-Apr-00):     else:  ;any other key press does nothing
1.9          (schlegel 14-Apr-00): endcase
```

This shows the version number in which each line last changed, the date, and the user who committed the change. This is an excellent way to track down bugs and blame them on someone.

It is useful to examine the history of a file (command = log, NOT history), as in the following example:

```
> cvs log atv.pro
RCS file: /usr/local/cvsroot/idlutils/pro/plot/atv.pro,v
Working file: atv.pro
head: 1.26
branch:
locks: strict
access list:
symbolic names:
        v5_3_0: 1.26
        v5_2_0: 1.26
        v5_1_7: 1.26
...
        v1_0: 1.7
keyword substitution: kv
total revisions: 26;    selected revisions: 26
description:
----------------------------
revision 1.26
date: 2006/08/08 17:20:16;  author: blanton;  state: Exp;  lines: +32 -2
add mark
----------------------------
revision 1.25
date: 2006/08/04 16:50:00;  author: blanton;  state: Exp;  lines: +63 -1
add m and k keys to mark and delete points
----------------------------
...
```

```
----------------------------
revision 1.17
date: 2003/06/24 20:45:55;  author: dfink;  state: Exp;  lines: +45 -39
splot / soplot stuff back in
----------------------------
revision 1.1
date: 1999/07/09 20:33:18;  author: schlegel;  state: Exp;
Add plot directory.
=============================================================================
```

When one is finished with a particular module, and all changes are committed, the module may be deleted with

```
$ cd ..
$ cvs release -d idlutils
You have [0] altered files in this repository.
Are you sure you want to release (and delete) directory 'idlutils': y
```

This will remove the directory idlutils and all directories within it. Don't worry, all the files are safe and sound in the repository. You can also just delete the usual way, but CVS is keeping track of who has what checked out (but you don't care).

Tagging a version

In the course of code development it is convenient to assign a "tag" to the current state of the repository. Tag names can be any reasonable string, but often have a format like "v_5_3_12" where we use an underscore for tag names to differentiate from the periods used by cvs for revision numbers. Note that you never have to tag anything for cvs to be useful, but it is good form to tag e.g. a software pipeline before each run through your data, or a journal article before each (re)submission to the editor. The command is

```
cvs rtag v5_3_0 idlutils
cvs ci -m 'OOPS! forgot to make this one change' pro/fits/mrdfits.pro
cvs rtag -F v5_3_0 idlutils
```

where the second cvs rtag command "Forces" a retagging (also known as "moving" a tag). Once you have tagged, you can use the tag name anywhere you would use the revision number in a cvs command, e.g. to see what has changed in mrdfits.pro since v4_9_12

```
cvs diff -r v4_9_12 mrdfits.pro
cvs log mrdfits.pro
```

The log command will list all tags and which revision number they correspond to for this file. Don't worry about tagging too often; you won't run out of tag names.

Exporting a tagged version

It may be convenient to "export" a tagged version of your code to run, so that can be running while you continue to make modifications in your copy. Go to the directory where you want the code (perhaps a directory named for the version)

```
mkdir v5_3_0; cd v5_3_0
cvs export -r v5_3_0 idlutils
```

And that version will be pulled from the repository. Two nice things about exports - there are no CVS directories all over the place, and also the string "$Name: $" is replaced by the tag name anywhere it appears in any file in the repository. This is very convenient for automatically inserting the tag name into your output files, etc., so that you can always tell which version you ran.

Environment Variables

The behavior of CVS may be modified by using a number of UNIX environment variables. I have the following lines in my .cshrc file:

```
setenv CVSROOT :local:/home/dfink/CVS_repository
setenv CVSEDITOR emacs
setenv CVSIGNORE '*~'
```

CVSROOT points to the location of the CVS repository, where the files and all of their previous versions reside. The default editor launched by cvs commit is specified by CVSEDITOR. Filespecs of files you want CVS to ignore are listed in CVSIGNORE. I do not want CVS to pay attention to files ending with $\sim$ because these are backup files made by EMACS. EMACS does *not* make backup files of files under CVS control, but may have made one before you added the file.

To access a repository on a remote machine, type (for example)

```
setenv CVS_RSH ssh
setenv CVSROOT username@sdsscvs.astro.princeton.edu:/usr/local/cvsroot
```

where "username" is the username on the remote system. If the username is the same on both systems, it may be omitted. The CVS_RSH line forces cvs to connect using ssh (secure shell) instead of the default rsh. Most UNIX systems now require this.

If you wish to check out a cvs product from a different repository than $CVSROOT, it is possible to override with, e.g.

```
cvs -d howdy.physics.nyu.edu:/usr/local/cvsroot checkout hoggpt
```

Using a pserver

In some cases, software is provided to the general public via a CVS "pserver" or public server. You do not need an account on the remote machine to access the repository, but access is usually read-only. To download the idlutils repository from Princeton, type

```
cvs -d :pserver:anonymous@sdsscvs.astro.princeton.edu:/usr/local/cvsroot login
cvs -d :pserver:anonymous@sdsscvs.astro.princeton.edu:/usr/local/cvsroot co idlutils
```

The "login" line only needs to be done once for each repository you intend to use. There is a password for access, but in the case of idlutils the password is an empty string. "co" is an abbreviation for checkout. To set up a pserver, follow the instructions on the CVS web page.

Using CVS with EMACS

It is no surprise that version control is fully integrated with EMACS. The people who write EMACS undoubtedly use version control to keep organized.

EMACS automatically determines what type of version control software you are using and displays, for example, IDL CVS-1.1 on the information line near the bottom of the EMACS window.

You may use the Version Control menu under the Tools menu to invoke CVS, or type keystroke shortcuts. The most useful commands are

```
C-x v v  - commit
C-x v u  - revert the last version
C-x v l  - show history
C-x v =  - compare with last version (context diff)
C-x v g  - annotate
```

When appropriate, these commands cause EMACS window to split in two, and show the history, annotation, etc., in the bottom window. Just point at the top window and type C-x 1 to get back to a single window.

You will find that you seldom have to invoke CVS from the UNIX prompt if you work in EMACS.

Setting up the repository

The above example assumed that a repository had already been created and the module idlutils had been checked in. In this section we describe how to do that. This will be the central holding place for CVS files, so put it on a disk with enough space.

Make a directory called (for example) CVS_repository and make it writable by everyone with:

```
mkdir CVS_repository
chmod 777 CVS_repository  (or maybe chmod 2775 with group owner)
cvs init
```

Now you can start putting things in (importing) to the repository. (Make sure you have set $CVSROOT first!) The command cvs init causes the CVS_repository/CVSROOT/ to be created. This directory contains the internal CVS files that control everything.

Suppose you want to import your /pro directory. You would type

```
cd ~/pro
> cvs import -m 'IDL procedure files' pro dfink start
N pro/test.pro             - N means new file
N pro/prob3.pro
I pro/test.pro~            - I means ignored this file
I pro/prob3.pro~
I pro/#readfits.pro#

No conflicts created by this import
```

Now the files have been copied into the repository. In this example, "pro" is the directory name for the files within the CVS_repository, "dfink" is the vendor, and "start" is the releasetag. Don't worry what these are for now - at the level we are using CVS, these parameters are never used.

Now the module "pro" is ready to be checked out. To avoid overwriting the old pro directory make a directory called test and go there to check it out:

```
> cd test
> cvs checkout pro
cvs checkout: Updating pro
U pro/prob3.pro
U pro/test.pro
```

If the files look fine, you can get rid of your original copy of pro.

Mergers

Suppose you edit test.pro and remove a few lines. Meanwhile, someone else has changed the file also!

```
> cvs commit -m 'removed some stuff' test.pro
cvs commit: Up-to-date check failed for `test.pro'
cvs [commit aborted]: correct above errors first!

> cvs upd
cvs update: Updating .
RCS file: /home/dfink/CVS_repository/pro/test.pro,v
retrieving revision 1.1.1.1
retrieving revision 1.2
Merging differences between 1.1.1.1 and 1.2 into test.pro
M test.pro
```

In this example, changes were made in different parts of the file, and it is clear how CVS should merge the changes. When CVS does not know what to do, there is a conflict.

Conflict merge

```
> cvs update
cvs update: Updating .
RCS file: /home/dfink/CVS_repository/pro/test.pro,v
retrieving revision 1.2
retrieving revision 1.4
Merging differences between 1.2 and 1.4 into test.pro
rcsmerge: warning: conflicts during merge
cvs update: conflicts found in test.pro
C test.pro                    - C means look at this immediately!

emacs test.pro    and you will see
<<<<<<< test.pro
```

```
I am going to add all kinds of junk here
=======
I am adding all kinds of junk here - here is a little bit more
>>>>>>> 1.4
```

There is no way CVS can sort this out, so you have to.

Other tips

Each CVS command has a large number of variants, but I on a day to day basis I the commands I always need are:

```
cvs -qn up            (quietly check what would be updated without doing so)
cvs -q up -d          (quietly update including new directories)
cvs up -r 1.12 foo    (get version 1.12 of foo, and set sticky bit)
cvs up -A             (update clearing sticky bit, resetting to current
                        version of foo -- otherwise update has no effect)
cvs up -p -r 1.12 foo > foo.old    (copy version 1.12 of foo to
                        foo.old without setting sticky bit)
cvs diff foo          (see what changes I have made since my last update)
cvs diff -D now foo   (diff foo with CURRENT state of repository)
cvs ci -m 'message' foo   (check in new revision of foo)
cvs log foo           (check which revisions correspond to which tags, see
                         check-in history with all the messages)
```