IVILab Manifesto [1]

Written by IVILab members throughout the ages including Salena Ashton, Kobus Barnard, Jinyan Guan, Clay Morrison, Andrew Predoehl, Adarsh Pyarelal, Kyle Simek, Ranjini Swaminathan, and Manujinda Wathugala.

Last updated: February 12, 2022

[1] Document source: `svn+ssh://vision.cs.arizona.edu/home/svn/ivilab/manifesto`

Contents

Preamble

Welcome to the IVILAB manifesto **project** where we are gathering a substantive collection of information about the IVILAB and how we work together to the benefit of all. In addition to aggregating documents so that they are easy to find and easier to update, the plan is to add information that is not written down, including some of the history that explains why things are in their current state, and better articulation of guiding principles.

This document was born in late May 2017, and currently needs a lot of material and is very rough. You are invited to help. If you do help in a substantive way, please add your name to the author list.

Because the information about about how to get the latest version of this document is in this document itself (specifically on this page), you may have received a bootstrapping copy via email. However, you should strive to update your copy when you consult it for the first time in a while.

The web version of this document is located at

```
ivilab.org/private/manifesto.pdf
```

It is updated automatically as described below. The source of this document is in

```
svn+ssh://vision.cs.arizona.edu/home/svn/ivilab/manifesto
```

If you commit changes, then an SVN hook will attempt to build the PDF in `/home/ivilab/manifesto` on the machine "vision" and copy the resulting PDF (manifesto.pdf) to `/home/ivilab/public_html/private`, again on "vision". If your commit hangs, or the web version of the PDF does not get updated, then manifesto might not be building on vision, perhaps because you have not checked that it builds, or you have not committed everything including any changes to `texinputs`, or you have not accounted for browser caching, or a perhaps there is bug in the scripts that do this. If you suspect the last reason, contact Kobus.

Chapter 1

The Commandments

1. Thou shalt reply to email within 24 hours either by typing or using thy sacred auto-responder.
2. Thou shalt frequently commit thy work to the holy SVN repository.
3. Thou shalt illuminate (comment) thine incantations (code) clearly and concisely, always.

   > *Always write comments so that someone not acquainted with your project can easily understand what your code is doing (and why). It only take a couple of weeks before that someone is you.*

4. Thou shalt test the instructions thou writest by cutting and pasting the commands into a fresh instance of the environment, whilst pretending to know nothing about the process. Heaven awaiteth those who testeth their incantations with multiple religions (operating systems).
5. Thou shalt respect thine user's cognitive load, even if thine user is thyself.

   > *This relates to the principle of least astonishment. The user can cope with the unexpected, but they prefer it when things just work as they expect. This includes a notion that the program should do the things that the user knows the program can do, such as explain why it is failing. A second example, is*

a set of procedures that the user needs to do that clearly could be automated. The user might be astonished that they need to transfer data from a location that is known in advance (as it is in the instructions).

6. Thou shalt continuously ask thyself, "How doth this code fail and protect thine users from such plagues?"

7. Thou shalt write documents using either LaTeX or Microsoft Word augmented with Mathtype.

   The Microsoft equation editor is of the devil and OpenOffice is not sufficiently compatible.

8. Thou shalt have results informative of whether work is ready for publication two weeks before the paper deadline.

9. Thou shalt not ignore what is already in place without consulting the all mighty church (IVILAB group).

   What is in place is often broken, and you are expected to improve it. But doing a "better" job on your own local environment leaves the rest of the lab with a broken version. In addition, it is often the case that your fix has already been tried and has some disadvantages. Finally, while some of the lab code appears to reinvent the wheel, some of that code is 25 years old and predates many wheels. Nonetheless, often it should be replaced, but usually in such a way that the current way can be a fall back (see XXX).

10. Thou shalt use a sacred spell checker.

    XXX Other fluff related issues?

11. Thou shalt add additional commandments until we reach a count that is a holy power of two.

    There will be more commandments.

Chapter 2

Getting Started

IVILAB students are expected to update a lab notebook regularly and weekly, at the least. If you are new, your immediate tasks are to set up this notebook and be able to update it. This process may involve learning new tools and prerequisites. Read through this chapter, then begin to play with the system as soon as possible. Use good judgment about the depth at which you learn each new prerequisite.
Your mission is to set up and regularly update your notebook as soon as possible. You will find several references to sections of chapter 4 within this chapter. If you do not fully understand a particular prerequisite, read through the chapter 4 section referred to then continue reading this chapter.

The prerequisites are:

- Logging onto **IVILAB computers**. The very first step is to log onto the IVILAB computers and change your password on each one of them. This is discussed in more detail in § 4.1.
- Use **SSH keys**. This makes using SVN (next prerequisite) much easier. SSH keys are discussed in § 4.3.
- Use Subversion (**SVN**). See § 4.5.
- **Build the PDF** of your notebook from your LaTeX source on a regular basis. You can write your notes using an IVILAB computer if you like, but it is easiest to build on the same machine you write. In either case, you need to ensure that LaTeX is available by installing on your laptop. See appendix for details § A.1.7.

- Set up your **command line environment**. You will need to tell LaTeX where to find files by setting up the environment variable `TEXINPUTS` in your login scripts. If you are using a Linux, the default shell is `bash` and you can achieve this by editing your `!~/.bashrc` file. If you are using macOS Catalina, your default shell is `zsh`. Change this to `bash` by entering:

  ```
  chsh -s /bin/bash
  ```

  into your terminal command line. You can then edit your `./bash_profile` script. (See below for a bit more about TEXINPUTS).

Additional considerations for getting started:

- Use an efficient text editor. Undoubtedly, you already know how to edit text documents. If you have doubts about your efficiency, it might be a good investment to learn a powerful programming editor such as Vim, which most IVILAB folks use.

  If you are learning an editor, tell us about how it is going in your notebook!

- Know the Linux command line. Be aware that most tasks and processes, including the updating of notebooks, use the Linux (or Mac) command line. If you are unfamiliar with the command line, you may want to think about a strategy for learning about both the command line and shell scripting. Consider going through the swcarpentry Unix Shell online tutorial or the book *The Linux Command Line* by William Shotts. Begin with Digital Ocean's SSH keys tutorial.

  Again, if this is something you are doing, tell us about how it is going in your notebook.

2.1 Setting up Your Electronic Notebook

- We are switching from semester notebooks, to reverse chronological notebooks that need to have the current semester in them, but can go as far back in the past as you like.

- We are changing the preferred notebook location to ensure notebook privacy. Only the writer and their mentors will have access to them, so we need reconfigure where they live on the Subversion (SVN) server .

  The new notebook locations are backwards compatible with the previous locations so you do not need to scramble to switch.

- When you are ready to switch, you need to remove the old version from SVN. If you are starting fresh, you can simply remove it, or make a directory `$USER` in

  ```
  svn+ssh://vision.cs.arizona.edu/home/svn/ivilab/reports/ARCHIVE
  ```

  to keep an archival version (the SVN repository will have a copy of all previous versions regardless). If you are continuing with your current notebook, then you could save a copy on your computer, delete it from SVN, and use the copy as a starting point for the new one. However, the new one that you construct will be named differently, and be in reverse chronological order.

The first step is to setup the directory structure for ivilab related tasks, including your notebook. You can do this on your own machine (easy if it is some flavor of Unix such as linux or mac OS) or on one of the IVILAB machines other than vision. You can do both if you like. The following instructions are for a particular choice of directory structure. If you choose a different one, you will have to make some small adjustments.
In your home directory, or elsewhere if you prefer, create an **ivilab** directory, then go into it. You will need to be signed into lectura. [1]

Be sure to tell SSH that “vision" means “vision.cs.arizona.edu" in your SSH configuration file. Otherwise, the following commands in this chapter will not work[2].

```
mkdir ivilab
cd ivilab
```

Inside this **ivilab** directory, checkout the texinputs, templates, and scripts directories.

[1] If you do not understand what this means, stop reading here. Jump to § 4.1. If you do not have access to lectura, your first step from here is to email Kobus directly.

[2] If this doesn't make sense to you yet, please see § 4.3

```
svn co svn+ssh://vision/home/svn/ivilab/templates templates
svn co svn+ssh://vision/home/svn/ivilab/texinputs texinputs
```

Although not needed right away, this is a good time to checkout the manifesto and the ivilab script directories as well.

```
svn co svn+ssh://vision/home/svn/ivilab/scripts scripts
svn co svn+ssh://vision/home/svn/ivilab/manifesto manifesto
```

You should add the path for "scripts" to your PATH environment variable (see § 4.1.2 if you don't understand this yet).
Now make a directory in your private area on the SVN server for your reports. You can use any organization you like, but for concreteness, we will proceed assuming that you will want an ivilab subdirectory, with a reports directory inside that.

```
svn mkdir --parents svn+ssh://vision/home/svn/users/$USER/ivilab/reports
```

`$USER` is your login id on ivilab computers. If you have the same login id on the machine that you are using, and your shell (e.g., bash) is behaving well, you should be able to cut and paste that command as is.
Still in the directory `ivilab` check that the SVN directory you just created. You can call it what you like, but it is often best to mirror directory structures where applicable, and so for this example we will also use the name reports.

```
svn co svn+ssh://vision/home/svn/users/$USER/ivilab/reports reports
```

Copy the example notebook in `templates` to `reports`, but change "example" to your name. This is your notebook.

```
cd reports
cp ../templates/example-notebook.tex NAME-notebook.tex
```

Here NAME is your name in any form you like (e.g., login id, first name if it is reasonably unique, or FIRST-LAST).
Add your notebook to SVN.

```
svn add NAME-notebook.tex
```

Check that you can build the PDF using LaTeX. On macOS and Linux machines, providing that LaTeX has been installed, one way is to invoke the following command in the Manifesto directory.

```
make
```

However, you need to ensure that LaTeX can find the files in the "texinputs" directory you have checked out. On the mac and linux, the easiest way is to add the path to that directory into the environment variable, `TEXINPUTS`, which one would normally do in via their shell startup scripts (e.g., .bashrc). `TEXINPUTS` works much like the `PATH` environment variable in that multiple directories are separated by colons. You need to terminate your `TEXINPUTS` string with a colon, which tells LaTeX to look in the default place which was set up when you installed LaTeX. If that all seems mysterious, try the following in your `.bashrc` file:

```
export TEXINPUTS=".:$HOME/ivilab/texinputs:"
```

Of course, if you put inputs to LaTeX in some other place, you will need to modify the above to reflect that. If you have other places you want LaTeX to look for files, then you should add them to the colon separated list.
Once your notebook builds, and you have checked the PDF, update your notebook in the SVN repository:

```
svn commit
```

The above will ask you to provide information about the changes. This is often important, but perhaps not for notebooks. You can circumvent doing so by the following variants which you might eventually make a shell alias for.

You may also want to create an alias for accessing and building your labbook. For example, in your bash file, consider adding:

```
alias notebook='cd ~/ivilab/reports; vim NAME-notebook.tex'
alias build='pdflatex NAME-notebook.tex; open NAME-notebook.pdf'
```

Be sure to check for existing aliasas before creating another. For details about learning SVN, see § 4.5.

```
svn commit -m ''
svn commit -m 'Lazy commit'
```

The Last Initialization Step. Now you need to add your notebook URL into a file in a shared IVILAB directory. To get your notebook URL, you can into your notebook directory and type `svn info`.

To change the shared record of notebook URLS, you next need to get the directory it lives in onto your system via SVN. In our example directory structure, a reasonable place to put this local copy of that directory is in `~/ivilab`. (For those new to Unix, `~` is interpreted by the shell as your home directory). To proceed:

```
cd ~/ivilab
svn co svn+ssh://vision/home/svn/ivilab/externals externals
cd externals
```

Now you need to edit the notebook-externals. Read the instructions inside this file, add a line for your notebook directory, then commit your change. Once you have done that, email Kobus that you have done so.

2.2 Updating Your Electronic Notebook

Notebook entries should have headings that specify the date of entry (there is an example in the template to get you started). Your notebook entries should be in reverse chronological order, so that the most recent entry is first (new for 2019). You should update your notebook relatively often (at least once a week). If you are just getting started, your first entry might be about getting to the point of being able to add notebook text (i.e., `Hello World`). If you are away for an extended period of time, or have no content for some reason (GRE exam?), simply make a short (dated) entry explaining that. This distinguishes not being able to do it from alternative explanations such as forgetting about it, not having interest in the process. Finally, your main recent writing activity might be some other document that lives in SVN, In this case, your notebook entry can simply say that you updated that document substantively, with perhaps a sentence or two about what the main improvements were.

Once set up, the work flow is to update your notebook (frequently), build the PDF and proof read it, improve it, rebuild, improve, rebuild, and then commit the source (the .tex file). Do **not** add the PDF or auxiliary files to SVN.

Figures, tables, and references are welcome (you need to add the images to SVN). If you are working with images, more often than not, you should include some in your notebook. The notebook can also refer to code or the location of other documents that have been updated. If you are not

actively writing a paper for a particular venue, you can view the notebook as gathering bits and pieces of the story and results for papers, thesis, or longer reports that are yet to come. Finally, to the extent that you **are** actively writing a document, you can simply note major updates in your notebook and provide a pointer to the updated document (usually an SVN location).

2.3 Please Commit to Keeping a Good Notebook.

Two or three bullet points is not a lot to say for a week of research. Typically you will have several paragraphs, and up to several pages if you have figures and tables, which should often be the case. What you write need not be completely tied to research, which is often the case when you are just starting out. It is fine to report on what you learned as part of a reading group or a study group or self study inspired by what is going on in the lab. Typically working with the IVILAB is a 9-10 hour a week commitment for undergraduates, so it is reasonable to spend several hours reflecting on what is going on and writing it down. Writing is a record of the thought process, and helps you to be clear about what you are thinking. If you are unsure what to write, then the following suggestions might help:

- Your understanding of your project and how that relates to the small tasks you have done recently or are planning on.
- What you did in enough detail that the reader is not wondering about what was done. Reread (or better yet, have your buddy read) what you have wrote, and ask what details are missing. For example, "it worked better" is not very informative. What was the data? What was the testing protocol (e.g., train vs test splits)? What were the error measures? Is it better beyond a doubt? Do you really believe that it is better? Are you willing to bet money? You may not be sure, but you can be sure about what you did. Write it down!
- What setbacks did you encounter? If you are coding, what was your worst bug? What would you do differently?
- What are the next logical steps? Why? Have you considered alternatives?

Finally, you should refer to chapter 7 for suggestions on how to improve your writing. If you treat updating your notebook as simply a chore, then you will not learn important skills, you will be bored, and your notebook will be boring.

Chapter 3

Basic Infrastructure

3.1 Mail lists

We use the U. Arizona (Sympa based) mail list system to send mail to various sets of people. You can add or remove yourself from most of the lists, or you can ask one of the list administrators (listed below) to add or remove you. These lists are generally set up so that their existence is not easy to discover if you are from the outside, but that anyone can send to the list. (If we get too much spam we might change this). However, some lists were originally setup so that only list members can send to the list, which is problematic if you want to use a different email address when you reply to a list message than the one that you were signed up with. A few of the lists may need to be changed to make it so that anyone can send to them (subject to not having spam issues).

In addition to asking one of the mail list administrators, you can interact with the mail list system in two ways. The first is via emails containing simple commands. The second is via the mail list web interface (`https://list.arizona.edu`[1]) which provides additional capabilities if you log in using your UA netid.

3.1.1 Mail list administrators

Kobus Barnard (kobus@cs.arizona.edu)
Clayton Morrison (claytonm@email.arizona.edu)

[1] as of 20170529

Adarsh Pyarelal (adarsh.pyarelal@gmail.com)
Chinmai Basavaraj (chinmaib@email.arizona.edu)

3.1.2 Mail lists and their uses

Mail lists are case insensitive, so fancy use of case (e.g., CompTIES) is just for clarity.

ivilab-colloq@list.arizona.edu

Announcements of general interest to those in the community interested in IVILAB topics (computer vision, machine learning, and interdisciplinary applications). During terms that we hold a weekly seminar, the topics are announced via this list. This list has quite a few members that are not active in the IVILAB or at U. Arizona.

ivilab-support@list.arizona.edu

This list is for announcements, fault reports, questions, and answers regarding IVILAB hardware and software. Most active IVILAB people are on this list, and so you might get answers from one of any of a number of people, which is efficient.

ivilab-gpu@list.arizona.edu

This list is for announcements, fault reports, questions, and answers regarding using GPUs, either on the IVILAB machines, or IVILAB projects running on UA HPC.

ivilab-admin@list.arizona.edu

Use this list for problems likely needing help from someone with privileges on the system in question. It is approximately those people listed above as mail list administrators.

ivilab@list.arizona.edu

This list reaches core IVILAB members, but “core" is not well defined. It is a vague function of longevity, seniority, and time commitment. We have historically used this list largely to organize the IVILAB lab meeting, which is mostly about planning and infrastructure. All are welcome to attend, but some are expected to attend, and for others it might not make sense to attend. So the list has evolved to a blend of folks expected to attend the lab meeting and others who want to. In summary, the purpose and use of this list is not entirely defined and is UNDER CONSTRUCTION. Currently, it is not a list that you should join directly.

ivilab-summer@list.arizona.edu

This list is for announcements and discussion about IVILAB summer activities, which are typically mostly about the boot camp.

ivilab-assembly@list.arizona.edu

This list is for assembly (big mechanism) research.

ivilab-code@list.arizona.edu

This list is for topics about IVILAB code treated as an entity independent of any particular research activity. By contrast, code development for a research project would likely be discussed on a project oriented mail list, and issues using the IVILAB code, or breakages noted due to updates by other parties, would likely be reported to ivilab-support. However there is some overlap between lists, and it is usually not an issue to send to the incorrect one, as overlap in topics will imply overlap in subscribers.

ivilab-faces@list.arizona.edu

This list is for face understanding research.

ivilab-math@list.arizona.edu

This list is for mathematical topics of interest to IVILAB researchers. It was created for a math topics reading sub-group of IVILAB folks, and we keep it around for such reading groups, as well as a sensible place to share or query math topics.

ivilab-ugrad@list.arizona.edu

This list is for announcements and discussion specific to activities specific to undergraduate researchers. For example, we sometimes have special seminar series for undergraduates, and if so, they would be announced using this list. The usage of this list generally ebbs and flows with such formal activities, since, for the most part, undergraduates are included in all lab activities.

ivilab-web@list.arizona.edu

This list is for IVILAB web development. This includes web page development, as well as web servers infrastructure.

ICIbeer@list.arizona.edu

The IVILAB social activities mail list. It has been semi-dormant for a few years, but will likely see renewed duty starting fall 2017. (ICI stands for Interdisciplinary Computational Intelligence, as well being suggestive of “cold").

compvision@list.arizona.edu

This list has been redefined as being for any computer vision topic that its members choose to use it for. It originally played the role that ivilab-colloq now plays. This list might prove to be obsolete soon.

compvislab@list.arizona.edu

[OBSOLETE] This list has been superseded by “ivilab".

compvislab-SLIC@list.arizona.edu

[DORMANT, OBSOLETE] This list was for topics related to the SLIC project which is close to finished. If SLIC re-emerges as an IVILAB research topic, this list will be renamed to ivilab-SLIC.

compTIES@list.arizona.edu

This list is for the compTIES research project. It is called "compTIES" rather then "ivilab-compTIES" for historical reasons, and we anticipate and because compTIES is unique enough.

comp-si@list.arizona.edu

Thus list is for computational social intelligence broadly defined, and has many members outside of U. Arizona. It was created in conjunction with our NSF funded workshop, but has been relatively quite recently.

uqg@math.arizona.edu

This mail list is no longer handled through U. Arizona mail lists. Rather, it is a Google group. I do not know if it is possible to join directly

KOBUS SAYS: TODO: find out!, ←

but you can send mail to Kevin Lin (klin@math.arizona.edu) and he will add you.

cs-colloq@list.arizona.edu

This is a public list for announcements about U. Arizona computer science colloquia and similar topics.

3.2 Calendars

We have yet to find the perfect way to have good shared calendar capability. We have gone from running our own CalDAV server in the days when such things were not generally available, to (now) using Google Calendar as our CalDAV server. Google's calendar support is limited and draconian with respect to controlling who can do what, and hence it seems simplest to give permission to those who would like it on an individual basis. Also, Google sucks at user interface, so most of us subscribe to it from some other program (e.g., iCal).

If you have a Google account, Kobus can give you access to the calendars listed below, which are a mix of Kobus's calendars and truly shared calendars. Hence access is either read only (you cannot change when I teach) or RW (you can add your out of town dates) as makes sense. To get access, email Kobus with the calendars you want access to ("all" is fine), and your Google account (typically this is your gmail address). Once you have access, you should be able to see them using iCal if you select "Calendar" under "Gmail" under "Internet Accounts" in "System Preferences" on your mac.

- IVILAB
  For meetings that involve more than two IVILAB members
- Teaching
  Kobus's teaching related schedule
- Kobus
  Kobus's meetings that generally involve at most one IVILAB member
- Talks
  Talks of interest (when someone bothers to add them).
- Out of town
  When people are out of town (if they care to add it)
- Deadlines
  Paper deadlines and early warnings thereof

3.3 Web pages

The main ivilab web pages are reached from http://ivilab.org. This is implemented as a virtual host on the IVILAB web server (vision.cs.arizona.edu). The pages themselves are in /home/ivilab/public_html on that machine. However, they are under SVN control, and are best modified via SVN. The repository URL is `vision.cs.arizona.edu:/home/svn/www/ivilab`. Once you commit changes to the SVN repository, they are automatically propagated to the live location via a SVN hook. So, to edit the pages, first update your local copy, then make the edits, then commit, then check that your changes are live by going the relevant web page.

3.3.1 Personal web pages

There are many solutions for hosting web pages, but those who wish to use the IVILAB web server for this purpose are welcome to do so. If you want to set this up, make the directory `public_html` in your home directory on the machine `vision`. You need to permit `public_html` and everything underneath it that you want to make public. If you are not being selective, one way to do this is:

```
chmod -R o+rX public_html
```

You should have at least the file `index.html` under `public_html`. Once you are ready, let an IVILAB administrator know that you are ready to try to go public. That person will be able to make a link in `/var/www` pointing to your website. Your URL will be:

```
http://vision.arizona.edu/$USER
```

3.4 Compute and file servers

3.4.1 Server capabilities

We have two main compute servers gauss and laplace (and two legacy ones: v11 and bayes01) Laplace has two commodity GPUs and Gauss has two modern server GPUs (A100s). We run different flavors of linux on these machines to promote portable development, as well as being able to check against older code. All these three machines get their files from two mirrored

file servers. Finally, we also maintain a web server (vision), which is also our svn repo server.

3.4.2 UA HPC

Link to documentation: https://docs.hpc.arizona.edu.

3.5 Bibliographic tools

We need a better collective process for reference lists. Sorting this out is high priority for spring 2019. Hence, what follows will likely be out of date soon. However, we do need to carry on in the meantime, and what follows is the protocol that many of us have been using, which is likely easy to integrate into possible future systems [2] without too many problems (e.g., perhaps with scripts).

Suggested interim process. In the source directory for a document that has references, simply include one or more bibtex files (dot bib). There names should be unique to the IVILAB. Some possibilities in use so far are *project_ name*.bib (the references are focussed on a project) or *your_ name*.bib or *username*.bib if the references are a big collection maintained by you for many projects[3]. Since this will likely get changed to some centralized system, this is one place where a bit of hacking might be justified. For example, if you have two document directories that share a bibtex file, perhaps using symbolic links to a nearby file (i.e., in a parent or sibling directory) could make sense (in general, one should be careful using symbolic links within our SVN repositories). A different alternative is SVN externals for pointing multiple copies to some location.

Within the directory that contains the bibtex file, you should include PDFs for the cited papers. I suggest putting them in a sub-directory called "ref_PDF". Again, we will clean this up, once we have finalized a plan for doing so.

[2] Sneak preview. After revisiting a number of use cases, it seems that we should prioritize implementing a central reference data base as planned some time ago. But there is no perfect solution to this problem, and some issues with it need to be studied a bit more. A likely candidate for the repository is Zotero.

[3] The square brackets mean replace the generic term with an instance, as often used in unix man pages

It helps a lot to make the name of the PDF the same as the bibtex tag (label), and further to create tags somewhat consistently with others. For now, we use:

[first_author_last_name]-[YYYY]-[venue_abbreviation]-[extra]

where *[extra]* is a small number (say 0-3) of hyphen separated words that are salient to help us remember the paper. For example, `Barnard-2008-IJCV-eval`. Often we have been a bit lazy about the *extra*, especially for our own papers which we know well, but it is essential when the tag is not unique without it. Author names should be capitalized to make it easier to distinguish from tags where beginning with the author does not make sense. Sometimes the *first_author_last_name* needs more words for names that are essentially multiple words (e.g., for former IVILAB PhD student Luca del Pero CVPR 2011 paper, we use `Del-Pero-2011-CVPR`). Sensible variations are fine. For example, a data set or software package might not have a well defined first author or year or venue (e.g., `OpenFace`).

Where does bibtex information come from? Bibtex files usually come from the internet, or existing reference data bases using other systems that can export bibtex format (e.g., Zotero, Endnote). For example, Google scholar, reference tools available from UA library such as IEEE and ACM, and scholarly community sites such as `http://openaccess.thecvf.com` and `https://papers.nips.cc/` all can provide the bibtex information for the papers they provide. So, you usually do not need to type in all that information. However, you will find that the tags are not done consistently, so you will need to edit them. You will find that the tags from `http://openaccess.thecvf.com` are close to ours and probably easy to convert, but there is no consistency. A completely consistent system for all papers that have ever been published on all topics would lead to ugly tags that likely would include the DOI. So we need to experiment with our own system.

3.6 Lab equipment

UNDER CONSTRUCTION.

3.7 Books

UNDER CONSTRUCTION.

Chapter 4

IVILAB Computing

4.1 Accounts and Disk Space

Figure 4.1 shows a schematic overview several of the computer systems that will be the topic of this section.

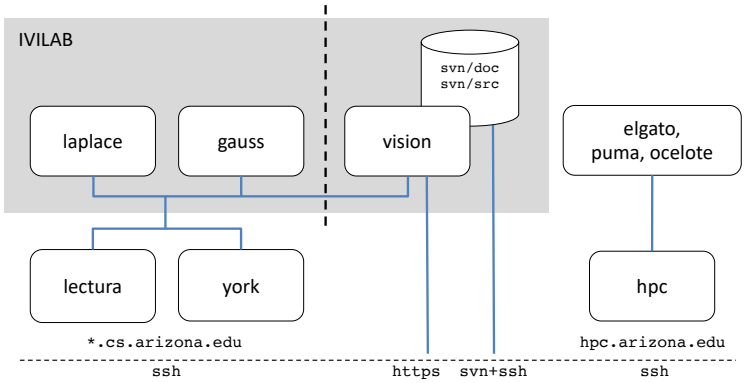

Figure 4.1: Schematic of IVILAB and related computer systems.

The IVILAB computers are part of the U. Arizona computer science (CS) network, and they are behind the CS firewall. Thus, to use the IVILAB computers you first need a computer science login. If you are a CS student or registered in a CS course, you should already have one. This is the login id you use to login to the machine "lectura.cs.arizona.edu". If you are completely new to using external machines, then you may need to learn about "ssh"

(§ 4.3) to check that you can login to lectura from your laptop. If you have forgotten your password, then visit:

```
http://www2.cs.arizona.edu/computing/services/
```

If you do not have a CS account, contact Kobus so he can request an account for you.

Assuming you have a CS login, Kobus can set up accounts and directories on the IVILAB machines[1].

Usually we use lectura (or "york" if lectura is not responding) to tunnel into the IVILAB compute servers, laplace, gauss (and legacy Bayes01 and v11), and the web server, vision. Until you have tunneling set up (see § 4.3), you can login to these machines by first ssh'ing to lectura, and then ssh'ing from lectura to one of these machines. Once tunneling is set up with ssh keys, the connection is still brokered by lectura, but this is all transparent. You can use lectura for basic purposes, but not for CPU intense research tasks. Generally we just use laplace or gauss, leaving v11 and Bayes01 for legacy testing.

You will have received initial passwords for those three machines from Kobus, along with the following information which is repeated here for completeness. First note that the machines do not share disks or passwords with lectura. The passwords for the cpu servers can be the same, and can be the same as for lectura (your choice). However, the password on "vision" should be different. Vision is a web server and we consider it less secure. So, if someone compromises vision, then they should *not* be able to compromise any other machines. This also means that you should *not* store ssh keys on vision that allow you to log onto a more secure machine. Login *to* vision from somewhere else using ssh keys is fine, and in fact you will want to set this up because vision is our SVN server (see § 4.3 for how to set up the keys, and § 4.5 for more about our SVN repository).

By default, your shell is the common default `bash`. If you want to use a different one without arranging it in your .bashrc file via the `exec` command, you can try to use the command `chsh`, and if you run into problems, get help from an IVILAB system administrator[2]. Unix shells are discussed further shortly§ 4.1.2

[1] As a reminder to Kobus from himself, or for anyone else who helps with administration, IVILAB account creation is handled via the script `add_ivilab_users`

[2] While `bash` has the largest following, which is a significant advantage, it might not be perfect enough to induce you to switch immediately if you are already are well versed in a different shell. Kobus, for example, uses `tcsh` despite documents entitled something like

Once accounts have been set up on the IVILAB machines, the following directories should be visible on the compute servers (e.g., laplace, gauss):

```
/work/$USER
/data/$USER
/space/$USER
/scratch/$USER
/backup/user_backup/$USER
```

Your home directories on the compute servers will be set to /work/$USER. At some point it the future, it might be possible to use the CS /home directories, but most of us prefer /work anyway as we back it up very aggressively and implement it with fast disks.

We try to keep usage on `/work` modest because we aggressively back it up. It is mostly meant for things that humans produce (code, documents). There is no strict guideline, but try to keep usage under 30GB for now. `/data` is for big data sets, `/space` is for storage that does not really need to be backed up (we try to anyway, but don't count on it), and `/scratch` is for fast storage that is not backed up (the build system tries to put object files there for faster compiles). You can use `/backup` how like – some find it useful, others have sufficient backup systems in place already.

Vision does *not* see any external disks. Your home will be under `/home`, and usage guidelines are less relevant. It is our web and svn server, so the main storage needs is driven by that, and backup is not as frequent as in `/work`.

Finally, when accounts are created, a user specific SVN repository is created. Its URL is:

```
svn+ssh://vision.cs.arizona.edu/home/svn/users/$USER
```

This can be used for multiple purposes, but at a minimum, it will house your notebook directory.

"tcsh considered harmful" written long after he learned `tcsh`. Technically, the holier than thou folks are probably mostly complaining about writing scripts in `tcsh`, and the shell you interact with need not be the shell you write scripts in. Having said that, perhaps unfortunately, a lot of the build system (§ 5.3) is largely composed of `tcsh` scripts. It is also unfortunate that `bash` syntax is so obscure and bug-ugly

4.1.1 Groups

Access to a number of items is controlled via the unix group "vision". For example, this is how the access to the SVN repository is controlled. You should be in this group on every IVILAB computer. We used to ensure sure that everyone is also in that group on the wider CS system, and we more or less do so, but currently nothing you will do needs this to be the case.

To check that you are part of this group on a particular machine, use the following command: cat /etc/group | grep vision

Access to change additional shared resources such as binaries on /misc is controlled through the group "visadmin". Membership to "visadmin" are provided on a need basis due to the small security risk in installing binaries that others will run. If you need to do a visadmin task only on rare occasion, it may be best just to ask someone in the group to do it.

Access to psychology data is controlled using the group "psych". This is sensitive data that you should NOT have access to unless:

```
1. You are part of the project.
2. You are IRB "certified".
```

Access to neuron data is controlled using the group "neuron". This is privileged data that you should NOT have access to unless:

```
1. You are part of the project.
```

This criteria is mentioned here simply so that errors in access are recognized as such. In other words, if you discover that you have access to psychology data, but are not part of the project, then please notify Kobus ASAP.

4.1.2 Linux and mac OS shell configuration

KOBUS SAYS: There is a bit of synchronization needed with § 4.6 ←

Our servers described above run linux, and most IVILAB folks also run linux or mac OS on a laptop. The mac OS is similar enough to linux that scripts can be shared with some bits and pieces of code being conditioned on

the system. In general, either you already know a fair bit about using unix like systems, or you will soon.

One makes their shell behave how they want through startup scripts, either by building them from scratch or by adapting one of many possible existing ones, either within the lab or beyond. If you choose to use the default "bash" shell, usually this means editing `.bashrc`. If you use some other shell, then you would edit the corresponding initialization file. Arguably, the fanciest startup scripts in the lab are Kobus's, but they are for tcsh. Kobus provides a few scripts to supply some of the features to bash users, and is willing to provide more[3]

A key setting in your initialization file is the `PATH` environment variable. At a minimum, it should include the two svn controlled directories. These should not be in your path until you check them out. See §2.1 to check them out first. (Note that a robust startup script only *adds* directories to PATH that exist).

To add to `PATH`, (/.bash_profile for macOS or /.bashrc for Linux):

```
export PATH=~/ivilab/scripts:"$PATH"
export PATH=~/ivilab/src/Make/scripts:"$PATH"
```

4.2 Shared file system structure

UNDER CONSTRUCTION.

File organization and naming principles. There is no perfect answer to organizing and naming things, and seeking it out would take too much time anyway. We would like reasonable systems that we improve over time in response to use cases. However, a key ingredient is **consistency**. This is a good principle in general, but it is critical in a system that shared among many. We are currently not consistent enough, and we need to push towards more consistency. A second principle is modularity, which amounts to dependencies being clear, or (even better, automatically accounted for). If a

[3] An example of a bash wrapper is the script src/Make/scripts/init_compile.sh that exposes many of the environment variables that we use to build code to the shell. An example of a wrapper that does not exist, but perhaps it should, is one to set the PATH environment variable.

system is consistent and modular, then it is easy to reconfigure it virtually via pointers, or non-virtually using scripts.

Organization structures tend to have two attributes in various proportions: Project based, and type of object (presentation, code, etc.) based. Both have advantages and disadvantages. The first has things associated with a project close to each other, which is sometimes convenient. The second promotes sharing across projects better. Currently we use both structures, but we are biased towards the second. We have had a long history of projects sharing code, and the single source directory has certain conveniences with respect to having version numbers be valid for all dependencies. However, some software development processes are now diverging from what we used to do most. As a second example, when I prepare presentations, I often like to use or review material from other recent presentations, since I am often making a larger point than simply presenting a specific topic.

Files and directories should have good names. This is ill-defined, but names should be relatively informative, relative unique, and not too long. For example, main.c is not informative or unique. Nor is lit_review.tex. It is clearly a literature review but of what? And it also clashes with your literature review of a second topic. Good naming takes work, and we should not be shy about fixing poor ones.

Shared file system. We maintain a shared file system structure for several reasons. First, to reduce the time we (especially Kobus), spend hunting for files, and waiting for email replies as to where they might be. This is especially important when the person who might know were the files are is not longer active, or traveling. It is even more important if one is concerned that the file might never have been in a place accessible to others who might need it. So, if you create a poster or a paper, and someone else (usually Kobus) wants to grab a figure from them for a talk or a research proposal, they need to be someplace where they can find it quickly.

A second reason is help ensure things are backed up. While one should not rely on the lab's backup processes, we need to have some confidence that we have multiple backups. Third, we want to promote sharing of code and other products such as references. If code is not in one place, then you will be more likely to duplicate it. Finally, organization and naming are important, and we all need to work on getting better at it.

The precise structure is meant to evolve over time based on discussion and observing use cases. One reason for documenting things is help this evolution. Most of the shared file system is in SVN, with the main exception

being data, which tends to be large and relatively static. Even if we retire the use of SVN (e.g., we become a cool lab that uses git), the structure will live on. Finally, how you subscribe to files that you participate working on is up to you. It is often easiest to mirror the structure in your own system or home directories, but this is not required. As long as others have the illusion of the agreed upon structure, there is not issue.

The main components, which correspond to main file server disks and/or SVN repositories are as follows:

- **Data**
  Data sets live in /data, which currently is largely symbolic links to /data_2, /data_3, and /data_4, merely because each of these disks are not big enough to hold everything. There are two parallel organizational systems — by username and by project or topic. Many data sets under project names simply symbolically link to the data in username directories. We might phase out the use of username directories, but this entails further thought and discussion, although where ever we use a symbolic link one could argue that we should move the data and reverse the link. It also seems that mirroring the structure in the project SVN repository when appropriate would be a good goal to converge to, but currently this has been done haphazardly.

- **src**
  Source code lives in the src SVN repository, which has subdirectories for build scripts (Make), libraries (lib), and projects (projects). We aim to have as much of the code in libraries as possible to promote sharing, but there will generally be non-negligible code that is specific to a project as well. A project directory often will have sub-directories for its own special library code, test programs, associated data preparation and/or visualization programs, associated scripts, and documentation. See XXX for more details. Where possible, it seems sensible to organize the src code for projects like the project SVN repository when appropriate, but so far this has not been a goal.

- **projects**
  We have a collection of SVN repositories for collections of related projects. Each repository contains directories containing shared documents, meta data that we construct (which entails version control), and

other miscellaneous project material. Currently the main repositories for groups of related projects are:

```
SLIC TIES assembly  astronomy faces fiber_bundle_SR fMRI tracking
```

and more will be added soon as we continue to organize and start up projects for the new academic year. Note that several of these have just been created, and may not have a lot of content.

Within the above categories, we typically have with sub-directories for more specific endeavours (e.g., "faces/blink", "faces/landmarks", "faces/-neonate_pain"). Within such directories there will generally be a "reports" directory. Most projects will end up with at least one document about the research, and one that is a literature review[4], that will go under reports. For example, within "faces/landmarks/reports" we have the two directories "landmark_detection_evaluation" and "landmarks_lit_review". The bit of redundancy in the name helps when we move files around, but is not absolutely necessary. However, the main latex file **must** have a unique name, not just "lit_review", otherwise the generated PDF will be called "lit_review.pdf" which is not informative enough.

Some of the specific projects contain directories that are feeds from github repositories. For example, the master copies of:

```
assembly/world_modelers/reports/conditional_probabilities
assembly/big_mechanism/reports/big_mechanism_assembly
```

directories in ml4ai github repositories. However, I do not necessarily recommend doing this unless you can write down the benefits that you expect. After due consider you feel that this how you want to proceed, then you can check out the assembly repository and look at the files named: `externals.txt`. However, this is not well tested yet, and perhaps best to talk to others about the latest lessons learned.

- **ivilab**
  An SVN repository that contains various elements related to lab administration and shared activities including:

[4] Sometimes combining these might make sense.

  - manifesto
    The source code for this document.
  - scripts
    Shared scripts other than the ones for building code.
  - reports/username
    This is where notebooks live (technically these live elsewhere and only a link lives here). Other reports written by only person could live here as well, but more likely a better place is in projects.

- **papers**
  An SVN repository that contains XXX
- **talks**
  An SVN repository that contains XXX
- **www**
  An SVN repository that contains XXX
- **users/$USER (semi-private)**
  An SVN repository that is specific to each user. It is generally private, except that Kobus and perhaps a few others have access. This is the default location of the lab notebooks, but can be used for other purposes as one sees fits.

4.3 The Secure Shell (SSH)

You are likely familiar with SSH, but you can still expect to `man ssh` quite a number of times in your IVILAB tenure. SSH versions are converging to the point that we can assume that you are likley using OpenSSH or a very similar implementation. SSH is nearly always installed and ready to go on Linux and Macs. Die hard Windows users will have to become experts at PuTTY, or if they use Windows 10, they can use a `bash` shell that is an optional feature. See here for instructions on how to set it up. The following are a few SSH issues that have confused newcomers to the IVILab in the past.

- The first time you use ssh to log onto a machine, or if the machine has been reconfigured in certain ways, or you have deleted your `~/.ssh/known_hosts` file, you will be prompted by a serious sounding challenge

about whether you really want to do this. Just type `yes` unless you suspect there really is an issue (rare).

- SSH is pretty good about helping you with X11, but you have to tell it to do so with the `-Y` flag. Also, you will need to allow your X11 server to allow other hosts to write to the display. The command to do this is `xhost +` which usually lives in a startup script). If you do not know what X11 is, you can probably ignore this (XXX — it is not yet clear whether we need to discuss X11 in this document),

4.4 Using SSH keys

Contributions from Kobus Barnard, Jinyan Guan, Clay Morrison, and Kyle Simek (via original Wiki page).

We use SSH extensively in the lab, so before long, you'll get tired of typing in your ssh password numerous times a day. More importantly, many tasks require one program or script to log into another machine, (e.g., a `cron` job)x. Other tasks are impossible to do unless you automatically log in through SSH without a password.

Luckily, with SSH keys, you have more significant control over how you access machines and improved security for a number of scenarios. Most of what you need to know about SSH is below, and/or in on-line tutorials. However, the definitive guide for clarifying any details or advance usage is: *The Secure Shell: The Definitive Guide* by Barrett, Silverman, and Byrnes.

> *Debugging SSH keys can be a bit tricky. There are a number of ways to do most things which adds to the confusion. You can use the `-v` flag to get some debugging information (more `-v`'s give more information). Pay attention to permissions including that of your home directory which should only be writable by yourself. Proceed methodically, don't try to get everything to work at once (always good advice), experiment, test, and use Google or ivilab-support@list.arizona.edu. A good way to debug SSH issues is to run the ssh* ***server*** *in debug mode ('-d' flag) in a separate window, but this requires administrative privileges on the machine you are trying to log into.*

4.4.1 Setting up SSH keys

At least 1 trillion tutorials for this exist on the internet (maybe more). Let's not reinvent the wheel; one good example can be found here. Again, the definitive guite is Barrett et al. Some comments about the steps in the linked tutorial:

- The passphrase you are asked to create is used to unlock your ssh key whenever the ssh client program needs access to your key (e.g., when you start a new ssh session). While strictly not required, we recommend that you create a passphrase for your key and remember it, at least until you understand the security issues below (XXX).

- Regarding the key pair that you generate: the *private key* is the file with the name *without* the '`.pub`' extension (default: '`id_rsa`') – this file should remain within your local `~/.ssh` directory and NOT be copied, moved, stored anywhere else. The *public key with* the `.pub` extension (default: '`id_rsa.pub`') is the only file that you will copy to any machine you want to connect to. In general, it is fine for anyone to see the public file – it poses no security risk. But the private key must remain protected.

- If you are on a Linux variant or MacOS, after creating the private/public key pair (using the `ssh-keygen` command) ensure that your .ssh directory has the correct permissions set. You can do this with the following command:

```
chmod 700 ~/.ssh
```

- Regarding copying *public* keys (tutorial step 3): the tutorial shows two ways to do this (using `ssh-copy-id`[5] or the more verbose (but generally more available) `cat`). Note that for setting up connection to the IVILAB machines, you will need to copy the public key to 3 different machines:

  1. `york.cs.arizona.edu`

[5] On the Mac, `ssh-copy-id` is not available just because you have `ssh`; If you use Homebrew, you can install `ssh-copy-id` with `$ brew install ssh-copy-id`. If you use MacPorts, the corresponding command is `port install ssh-copy-id`.

2. Either `gauss.cs.arizona.edu` or `laplace.cs.arizona.edu` (accounts are shared between these machines, so "copying the key to one" will make is available for logging into the other as well).
3. `vision.cs.arizona.edu` (This machine does not share accounts with the other IVILAB machines; the IVILAB code repository is hosted here, so everyone will need access to this machine.)

When doing this for the first time, you cannot yet "tunnel" from york to the other IVILAB machines, so you cannot yet use the `ssh-copy-id` or `cat` methods to simply copy the public key from your local machine "directly" to the IVILAB machines. Here is a recipe you can follow to make the copies (there are a number of ways you could do this, this is just one):

1. First, use the `ssh-copy-id` or `cat` method (as described in the linked tutorial) from your local machine to copy your public key into the `authorized_keys` file on york.
2. Copy your public key from your local machine to york:

   ```
   scp id_rsa.pub USERNAME@york.cs.arizona.edu:~/.ssh/.
   ```

   where you replace `USERNAME` with your account name on york. (This is somewhat redundant, as the `authorized_keys` file on york already contains the same information, but this makes it easier to repeat using the same `ssh-copy-id` or `cat` method FROM york to each of the other IVILAB machines in the next steps.

   At this point, you could jump to § 4.4.2. The next three steps are about copying your public keys to IVILAB machines, whose access requires tunnelling
3. Login to york and move into the `.ssh/` directory (this latter step can be done with: `cd ~/.ssh/`).
4. Now, use the `ssh-copy-id` or `cat` method (as described in the linked tutorial) **from york** to copy your public key into the `authorized_keys` file on any of the IVILAB machines; you'll repeat this step for one of `gauss` or `laplace`, and again for `vision`. (NOTE: because you are currently logged in to york and therefore on the CS network, you may no longer need to use the '`.cs.arizona.edu`' specification

for each IVILAB machine – you can just use the IVILAB machine name.)

5. Finally, after making the copies in step 4, you can now remove the redundant `id_rsa.pub` file in your `~/.ssh/` directory on york:

```
rm ~/.ssh/id_rsa.pub
```

You can now log off from york. (However, still keep your `id_rsa.pub` file in your `~/.ssh` directory on your local machine – you may want to use the key again to set up connection to another machine, later.)

Even after doing the above, you cannot yet "tunnel" into the IVILAB machines from your local machine. To do this, you need to complete the steps in § 4.4.3 below. Until we do that, let's set our goal to log into york (or lectura) without typing our password, and controlling how often we type in our key passphrase as discussed next (§ 4.4.2).

- You do not need to do optional step 4 in the referenced tutorial.

4.4.2 Using `ssh-agent` to manage your keys

Substituting typing a passphrase instead of a password might does not immediately seem helpful. However, the standard protocol is to use `ssh-agent`, either implicitly or explicitly to manage the key so that the passphrase needs to be entered only once per work session. A work session might be a login into a Ubuntu desktop, or a login into a mac (which includes auto-login on boot if you have that set up). Having keys valid for a work session seems to be the most common security versus convenience trade-off.

Add the following line to the `~/.ssh/config` file on your local machine (create it if necessary):

```
ForwardAgent yes
Host vision laplace gauss
    ProxyCommand ssh USERNAME@lectura.cs.arizona.edu exec nc %h %p
```

Replace "USERNAME" with your actual username. If the username on your laptop is the same as on that on the ivilab servers you do not need to specify "USERNAME" at all. Now you can log into these machines directly from your laptop without explictly ssh-ing into lectura first then the machines listed.

After you set up your ssh keys (steps 3 to 5 in § 4.4.1 and § 4.4.2), you can connect to them from your local machine without typing your password. For the use just described, you do not need to have an .ssh/config file on lectura, but some uses might require that as well. There is much more that you might want to put into .ssh/config files on various machines. In particular, a strict analogy of the above does not on the UA HPC system. See § 4.4.5 below.

Ubuntu default desktop. On Ubuntu 16.04 (other versions need to be tested), you will only be asked for your passphrase when you start a new session (i.e., when you just logged into your laptop running Ubuntu). Then the authentication will last for the entire session until you logout. Therefore, you do not need to type your password nor your passphrase next time when you start a new ssh session as long as you do not logout of your computer. Specifically, you will see the following window (Figure 4.2) when you log into your laptop and issue an ssh command.

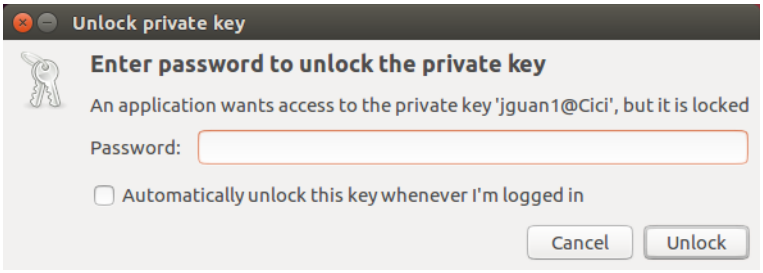

Figure 4.2: A prompt window when using nonempty passphrase for key authentication.

ssh-agent hacking. Another way to store the passphrase is directly via ssh-agent tool. (All methods use ssh-agent, it is simply the degree of exposure to you that is different). The ssh-agent tool stores unlocked ssh keys, so that only an initial entry of a passphrase is required for each key (often you just have one). It will be valid for the duration of a local login session. Here are some details of ssh-agent from Wikipedia (`https://en.wikipedia.org/wiki/Ssh-agent`)

> SSH is a protocol allowing secure remote login to a computer on a network using public-key cryptography. SSH client programs (such as ssh from OpenSSH) typically run for the duration of a remote login session and are configured to look for the user's private key in a file in the user's home directory (e.g., `~/.ssh/id_rsa`). For added security (for instance, against an attacker that can read any file on the local filesystem), it is common to store the private key in an encrypted form, where the encryption key is computed from a passphrase that the user has memorized. Because typing the passphrase can be tedious, many users would prefer to enter it just once per local login session. The most secure place to store the unencrypted key is in program memory, and in Unix-like operating systems, memory is normally associated with a process. A normal SSH client process cannot be used to store the unencrypted key because SSH client processes only last the duration of a remote login session. Therefore, users run a program called ssh-agent that runs the duration of a local login session, stores unencrypted keys in memory, and communicates with SSH clients using a Unix domain socket.

To use ssh-agent for your ssh key, run the following steps (see https://kb.iu.edu/d/aeww). (Depending on your system, only a subset of these steps are necessary. For example, on the Mac ... XXX).

1. At the prompt, enter: `eval `ssh-agent`` (Make sure you use the back-quote (`), located under the tilde (~) key, rather than the single quote (')).
2. Enter the command: `ssh-add [private-key-file]`, (if [private-key-file] is not provided, it will use the default file `~/.ssh/id_rsa`)
3. Enter the private key passphrase (the one you chose when you created the key.
4. To kill the authentication, run the command:

```
kill $SSH_AGENT_PID
```

The above assumes that ssh-agent is not running (it usually is), or you want to its process ID (PID) to be able to kill it as shown. Typically the

above commands would be entered while one is experimenting, and then the subset of commands needed to get the behaviour are put into alias or script or login script.

4.4.3 Tunneling using agent forwarding

Consider this scenario: you want to connect to laplace from your laptop, but you have to connect through lectura first, since laplace is behind the firewall. Since your *private* key doesn't exist on lectura (and should not be put there!), you'll be required to enter your ssh password to connect to laplace from lectura. There is a way for your key credentials to follow you onto lectura, so connecting to other machines from york won't require a password. When this is set up properly, then from your local machine you can act as though you are logging in directly to lectura, but behind the scenes the ssh client is handling connecting through lectura. Doing this is called *tunneling*.

Add the following lines to `~/.ssh/config` on your local machine (create it if necessary):

```
ForwardAgent yes
Host vision laplace gauss
    ProxyCommand ssh USERNAME@york.cs.arizona.edu exec nc %h %p
```

Replace "USERNAME" with your actual username. If the username on your laptop is the same as on the listed machines, you do not need to specify "USERNAME". Now you can log into those machines directly from your laptop without explicitly ssh-ing into lectura first. After you set up your ssh keys (steps 3 to 5 in § 4.4.1 and § 4.4.2), you can connect to the machines from your local machine without typing your password. For the use just described, you do not need to have an `~/.ssh/config` file on york, but some uses might require that as well. There is much more that you might want to put into `~/.ssh/config` files on various machines. See § 4.4.5 below.

4.4.4 Options for security versus convenience

Having keys that do not have passphrases (i.e., empty passphrases), or unlocked keys, allow you to login to other machines. The same is largely true for anyone who has compromised or stolen your computer. Hence one rule worth following is to never enable logging onto a more secure computer from a less secure one. A second rule is to always have backups because disaster

of some sort will eventually strike[6]. A third guideline is to be aware of what the consequences of a breach might be and adjust your protocols accordingly

It is tempting to legislate draconian protocols such as insisting that every key has passphrase. But an empty passphrase can make sense in some scenarios (it can even be necessary), and there are other ways to achieve the same ends. Security often comes at the price of some inconvenience, and trusting individuals with a job to do will tend to go for convenience. There is no simple solution or one size fits-all. Hence understanding the issues is necessary to make good choices. Some comments in this direction:

- While typing `ssh-add` once per work session might seem trivial, there are numerous ways that people use simplify things using shell scripts and aliases. These range from making registering with the agent wrapped into the first use of ssh (saves remembering to type `ssh-add`), to automatically prompting for a passphrase on the first login into a desktop environment, but then all subsequent terminal sessions (e.g., xterms) share the agent, to using keychains (mac). Interested persons should read up on such options.

- You can use multiple keys for different purposes. For example, a key that is used for automated backups or submitting commands to a compute server, but has no other power, might have (even need) an empty passphrase[7] By contrast, a key that can log onto machines as root might have especially strict handling[8].

- SSH allows for restricting logins to certain users from certain machines. One way to do this is within the `authorized_keys` file. If you know that you login into laplace only from york or lectura using only your username, you could make it so that someone with your key, but no

[6]For example, Kobus's laptop was stolen from his office a few years ago, and XXX

[7]For example, on the elgato cluster, you need an ssh key whose private key exists in the .ssh/ directory shared with the authorized_keys file, and, in fact, this is set up for automatically with the default login script. To emphasize previous points, this key should NOT be used for anything else, but such a setup is common, even necessary, in this kind of environment to make it so that any of the nodes in a cluster can login to another node in the cluster. Again, this key is only for that purpose, which is probably why the HPC admin folks set it up for you.

[8]Kobus uses a separate key for system administration, as it need not be unlocked very often, and thus there is less chance that some running off with his laptop will have that key loaded.

access to york or lectura would be stopped. (Of course, it is likely that entry is restricted to york and lectura anyway using `iptables`, but redundancy is fine here).

- Good security can be arranged for most scenarios with very little inconvenience, save for someone physically able to sit down at your computer as yourself. This includes someone who steals your computer, but does not reboot it. This inspires the practice of entering passwords after a certain amount of activity, etc. What is reasonable depends on your work scenarios. Use good judgement!

- Following up from the previous, this suggests that an alternative to passphrases and agents is keys with empty passphrases that are in encrypted directories or partitions that need to be unlocked for every login. This is currently Kobus's preferred method for some of his keys, but it is not necessarily more secure nor more convenient than other methods. The breach scenarios are similar, but he needs encrypted files for other purposes [9], so there is an overall saving in convenience.

4.4.5 Fancier SSH config files

There are a number of additional parameters you can set in the ssh config file (`man ssh_config`). These override the system supplied ones in `/etc/ssh/sshd_config` when permissible. You will put an ssh config file on most machines that you use as needed.

The following are excerpts from sample `~/.ssh/config` files, with some explanation of what the parameters do:

```
# ServerAliveInterval: Number of seconds that the client will wait
#                      before sending a null packet to the server
#                      (to keep the connection alive).
# ServerAliveCountMax: Number of seconds that the server will wait
#                      before sending a null packet to the client
#                      (to keep the connection alive).
#
#                      Setting a value of 0 (the default) will
```

[9] Among other things, the UA wants grades and other student information on laptops encrypted.

```
#                      disable these features so your connection
#                      could drop if it is idle for too long.

Host *
  ServerAliveInterval 10
  ServerAliveCountMax 3
  ForwardAgent yes

# The following line allows you to ssh to york using
# "ssh york" rather than "ssh york.cs.arizona.edu"
# If the USERNAME to york.cs.arizona.edu is the same as your
# machine USERNAME, you can omit the line "User USERNAME"

Host york
    User USERNAME
    HostName york.cs.arizona.edu

# The following allows you to log into vision via
# "ssh vision", "ssh vis" and "ssh vision.cs.arizona.edu" via
# york. Do not put ssh keys on york! Instead, make sure that your
# ssh config file has "ForwardAgent yes".

Host vision vis vision.cs.arizona.edu
    HostName vision.cs.arizona.edu
    ProxyCommand ssh york exec nc %h %p

# Similarly for laplace and guass. The "-e none" illustrated here
# is common in this context, but probably hardly ever matters.

Host laplace
    HostName laplace.cs.arizona.edu
    ProxyCommand ssh -e none york exec nc %h %p

# Notice that the incantations above do not work with UA HPC!
# The following does for now (we adjust to changes at UA HPC a lot).

Host elgato elgato-login.hpc.arizona.edu
    ProxyCommand ssh hpc.arizona.edu -W %h:%p
```

4.5 Using the SVN repository

We use subversion (SVN) for collaborative development of source code, and many, if not most, writing projects. The parts of SVN that you are required to know are relatively minimal and easy to learn. But it is critical that when working on an IVILAB project, you update the server side code frequently. The IVILAB has had much grief in the past due to version drift with respect to libraries, code that we cannot reconstruct, and code that only exists on one researcher's machine and cannot be realistically modified by anyone else. For those familiar with `git` and Github, SVN together with a shared server accomplishes roughly the same thing with modest differences. For those that are wondering why we do not use `git` like all the cool people, please read the discussion on this topic below (§ 4.5.15).

4.5.1 SVN intro

There are many tutorials and resources for SVN, including the definitive reference manual which is is available for download (`http://svnbook.red-bean.com/`). For completeness, we briefly introduce SVN here, which will help orient you for any needed self-study. (You will need to consult other resources to acquire sufficient expertise with this tool). We will follow the brief introduction by more detailed information on the IVILAB repository (§ 4.5.5) with working examples to get you started.

Subversion (aka SVN) is a type of version control system, also known as a concurrent versioning system. SVN solves a number of practical problems when developing code mentioned below (§ 4.5.2). To get started, you can think of SVN as a shared remote file system (AKA repository or repo) that keeps track of every "committed" version of all the files. As such, you will want to know how to copy, create, delete, and rename files on the system. Since it is a versioning system, you will also want to learn how to look at the history of a file or directory. More advanced use includes understanding conflicts which arise when two or more people edit files in incompatible ways. While this is sometimes necessary, and SVN provides reasonable tools to help you do so, it is still better to avoid conflicts by communicating with relevant parties over email about who should work on what when. (Example content of such emails might be "I am done with ssh.tex and I am locking svn.tex". In this context "done" implies *committed*.).

4.5.2 Version control benefits

KOBUS SAYS: This is some stuff from the Wiki. Perhaps not helpful, or should be shortened? ←

Code Repository. For many people, Svn is simply a centralized location to store a project codebase. It provides a simple command-line interface to download an entire codebase remotely, and to easily update the code when bug-fixes are released.

Incremental Backups. As well as storing the most current version of the codebase, all previous versions are stored as well. This is an essential tool when editing complex applications, because it allows you to edit one part of the code without worrying about breaking something somewhere else. If you discover weeks later that your “bug fix" broke the rarely-used foo module, you can roll back to the old version, or merge the old version with the new version.

Collaborative editing. Subversion allows multiple users to edit the same code without conflicts. If you’ve ever collaboratively worked on a project using ftp to download and upload code, you’ve probably experienced this scenario: You downloaded foo.cpp on Monday, and for the next 4 days, you wrote and debugged the most amazing code ever seen. Meanwhile, Jim was fixing bugs in foo.cpp. On Friday, you upload your spiffy new version of foo.cpp. But wait! 5 minutes later, Jim uploads his bug-fixed version of foo.cpp, completely overwriting all of your changes. If you were using svn, when Jim tried to upload foo.cpp, it would block it and tell him that his version of foo.cpp is out of date. Jim now has to download your new version of foo.cpp and merge his bug fixes into it before proceeding.

Tagging. A tag is like a snapshot of your project, frozen in time: you can continue to improve the project, but the tagged version will remain unchanged, in case someday you want to revisit this version. Subversion

allows you to tag a version of your code or documentation. For example, if you wrote a paper and the results came from v1.23 of your software, you might tag v1.23 as “CVPR Paper, Spring 2009." That way, when you refer to that paper later on, you can easily download and compile the version of the code that's relevant.

Branching/Merging. Subversion allows multiple parallel threads of development. Consider this scenario: You've just released Footron 3030 v1.0 to the public. Now you start working on drastic new features for v2.0. A few weeks into writing v2.0, you begin to receive bug reports about v1.0. What do you do? Do you fix the bugs in 2.0 and release v1.1 with a bunch of broken v2.0 features? That's no good. Do you split the project into two separate repositories, one for each version? Now every time you fix a bug in v1.0, you have to also fix the same bug in v2.0. Also not ideal. The best solution would be if you could branch off v1.0 and continue to develop it in parallel with v2.0, periodically merging the bug fixes into v2.0 as necessary. This is precisely what Subversion enables you to do. Usually, the "Stable" v1 code is stored in a "branch," while the v2 code is developed in the "trunk." When v2.0 is released, it becomes a branch, and the trunk is used for v3.0. A word of warning: branching and merging is one of the most elusive features for svn users, even experts. Branching is easy (it is just like tagging) but merging is difficult, so we don't use this feature often.

You can read more in Chapter 4 of the Subversion book, "Branching and Merging."

4.5.3 What goes into the repository

Generally we use the repository for human generated files, and do not include ones that are derived from those files. For example, the repository has code files, but not object files. As a second example, document repository directories may have `.tex` LaTeX files, but no `.aux` files, etc. Sometimes we include the `.pdf` result file for convenience under tight deadlines, but this is counter to the principle. However, figures and images are typically be included. These are binary files, but cannot be recreated from other files. However, there is gray area here. If a figure has a source file (let's say it is an OmniGraffle file) then the source file must be added (e.g., the `.graffle` file). Although theoretically sufficient, one might also add the EPS file because

others may not necessarily have OmniGraffle (and making the EPS might involve human selected choices).

4.5.4 File identifiers and permissions

Files in the subversion repository are referred to using a URL like identifier. If the repository is not local (the usual case) then the URL needs to identify the server name and how to access the files with respect to permissions. There are three possible ways to provide remote access. We mostly use `svn+ssh` which is showcased below. Alternatives include svn *server*, which we use to provide access to those who do not have credentials to ssh into our server (i.e., you, when you graduate) which is also described below, and through the Apache web server, which we do not currently use.

4.5.5 The IVILAB SVN repository

The IVILAB svn repositories lives on the machine `vision.cs.arizona.edu` in the directory `/home/svn`. Any directory below `/home/svn` (e.g., `/home/svn/src` or `/home/svn/users/kobus` that has approximately these directories

```
README.txt  conf/  db/  format  hooks/  locks/
```

is an svn repository. If you are on vision, you can refer to the repository "src" by:

```
    file:///home/svn/src
```

For example, to list the files you can do:

```
    svn list file:///home/svn/src
```

More likely, you want remote access via ssh. Assuming that you have setup ssh keys, and tunneling through the machine "york", the equivalent command is:

```
    svn list svn+ssh://vision.cs.arizona.edu/home/svn/src
```

Finally, you may want to access the repository without logging onto vision or using ssh keys. For example, you may want to access the svn repository from computers that we consider less secure than york and vision, or we may want to provide access to people who do not have login privileges (e.g., former affiliates or off site collaborators). There are two cases. First, read

only access might suffice. Then you can accomplish the same thing as above with:

```
svn list --username svn svn://vision.cs.arizona.edu/src
```

Notice that there are several differences in the URL. You will be prompted for a password if this is the first time you have done so. SVN might ask you if it is OK to store the password, and it is generally OK to agree (but note that the password might be stored in plain text on your system.) The password for a given repository can be found by logging onto vision, and looking in the file `conf/passwd` in the repository directory (e.g., `/home/svn/src`).

For write access, you need to specify a particular user name, and that needs to be set up in the files `conf/authz` and `conf/passwd` in the repository directory on vision. A particular user name is required because svn needs to keep track of who makes changes. We also use stronger passwords for write access, but they are located in the same place that the read only passwords are.

Trunk, branches, tags

The repository is just a file system, and we can put directories and files into it with any names we like. However, for code directories, a common convention is to have three directories: trunk, branches, and tags. What you normally would think of as the content of the directory is typically in "trunk". The other directories are in reserve for when we do tagging (sometimes) and branching (rarely).

4.5.6 Getting a copy of needed IVILAB code

To get started you will need the contents of at least four support directories. In addition, you might want a directory for a particular program you want to work on, or perhaps you will create a small program to explore things. We recommend that you put your mirror of the src code on SVN into `$HOME/src`, but this is not required. However, in what follows, we will sometimes use this assumption for illustration. On the assumption that you are in whichever directory you want to use for source code, the following commands will establish an SVN controlled copy of the four IVILAB code support directories ("co" is short for "checkout").

```
svn co svn+ssh://vision.cs.arizona.edu/home/svn/src/Make/trunk Make
```

```
svn co svn+ssh://vision.cs.arizona.edu/home/svn/src/lib/trunk lib
svn co svn+ssh://vision.cs.arizona.edu/home/svn/src/include_before/trunk include_before
svn co svn+ssh://vision.cs.arizona.edu/home/svn/src/include_after/trunk include_after
```

The last two are not used much, but the first two are critical. When you enter those commands, you will see quite a few files listed that have an “A" before their name. This says that the file was added. For completeness, let’s now suppose that you have been enlisted to work on a particular program, such as the “ties" program. You need to first be told, guess, or figure out, where the code is in the repository. It happens to be in src/projects/ties. It is perhaps simplest to mirror the directory structure, so you might want to put that code into `$HOME/src/projects`, which would entail making that directory, but you can put it anywhere you like. Assuming that you are ready to put “ties" into the working directory, you would use:

```
svn co svn+ssh://vision.cs.arizona.edu/home/svn/src/projects/ties/trunk ties
```

4.5.7 Updating

To update the code in the working directory (and beneath), you simply use `svn update` (or `svn up` for short). After doing any needed updates and telling you about it, svn will tell you the version that you are now at. To see updating in action, go to your `src/Make` directory, delete a file, and do `svn update`. You should see that svn restores the file for you. You can also specify a particular file to update.

4.5.8 Committing changes

You transfer your changes to files in your working directory (and beneath) to the repository using `svn commit` (or `svn ci` for short). You can also commit the changes to a particular file by providing the name of that file as an argument. You will find that SVN will ask you to log what you did by starting up an editor window (or you can put your message into the commit command using the “-m" option). It is a good habit to update before committing, as this makes dealing with conflicts easier(see section 4.5.10 for how to resolve conflicts in svn). (Say more XXX). In fact, if your copy is out of date, SVN will not allow you to commit your changes unless you update your working copy. A good practice is to always update before you making any changes, and always commit your changes as often as you can. For example,

you should commit your change as long as the code compiles or after some preliminary testing. If you are working with someone else and do not commit your changes for a long time, the chance that your copy will be out of date is relatively high since someone else might be also working on the same piece and has committed their changes before you do. Here are a few words of advice from “Dinosaurs" (Figure 4.3).

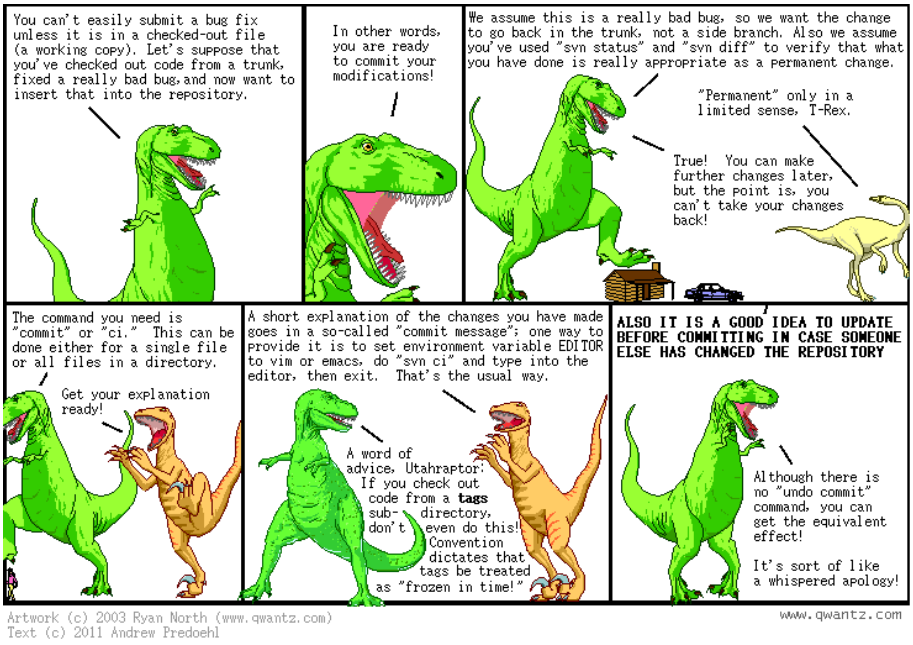

Figure 4.3: Dinosaurs teach you how to use `svn ci`

4.5.9 Adding and deleting files

To add a new file to version control, simply do

```
svn add <filename>
```

and to stop tracking a file and delete it, do

```
svn rm <filename>
```

4.5.10 Resolving Source Code Revision Conflicts in SVN

If two or more versions of one file are uploaded by two or more users and the same lines were edited, this will create a 'conflict' in the file. (Figure 4.4)

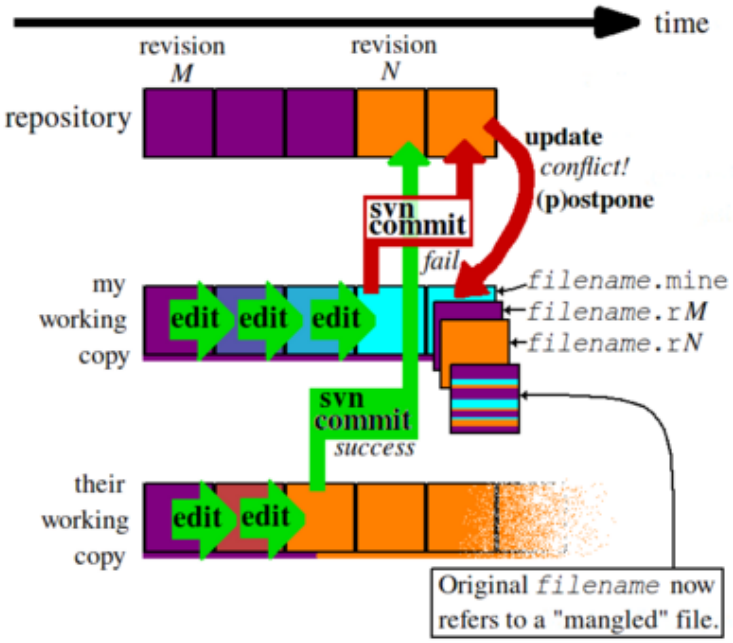

Figure 4.4: How conflicts occur and how Subversion enables you to resolve them

The file will be tagged with sections labeled either `"<<<<<<< .mine"` if the changes came from you, or `"<<<<<<< .rxxx"` if the revision is from someone else. The "x"'s represent the revision number. The two code blocks are separated by `"======="`. The multiple versions of each conflicted code block will be in the file, demarcated by these tags. To resolve conflict, after you choose a preferred outcome, you can run `svn resolve` with the following options:

1. `--accept mine_full filename` and then commit: Resolve all conflicted files with copies of the files as they stood immediately before you ran svn update (i.e., discards the changes in others)

2. `--accept theirs_full filename` and then commit: Resolve all conflicted files with copies of the files that were fetched from the server when you ran svn update (i.e., discards the changes that you make locally)

3. `--accept working filename` and then commit: Assuming that you've manually handled the conflict resolution, choose the version of the file as it currently stands in your working copy.

4. `--accept base` and then commit: Choose the file that was the BASE revision before you updated your working copy. That is, the file that you checked out before you made your latest edits.

For more information, see http://svnbook.red-bean.com/en/1.6/svn.ref.svn.c.resolve.html.

After the conflicts are resolved between programmers (Figure 4.5), the following commands should be used to reflect the resolution through svn:

```
svn resolved path/filename
```

4.5.11 Importing an existing directory into the repository

Adding a directory works if the added directory is a subdirectory of a svn controlled directory. But to add a completely new directory, such as a new program within, say, `src/programs/`, you use "`svn import`". This entails constructing a directory that contains exactly what you want to add. If you are putting an existing directory under svn control, you would typically move the existing directory to a safe place, create a new directory of the same name, create trunk, branches, and tags subdirectories beneath it, and then add the files that you want under svn control to trunk. For example, the old directory might have object files that should not be under svn control, but if the original directory was already exactly what you wanted to import, you could skip a few steps. You then add that directory to svn using `svn import`. For example, if your code is in `/work/yourname/src/foo` and it's part of a project that you want to call "superfoo," then you should do something like this:

```
cd /work/yourname/src/foo
svn import . svn+ssh://vision.cs.arizona.edu/home/svn/src/projects/superfoo/trunk
```

```
svn mkdir svn+ssh://vision.cs.arizona.edu/home/svn/src/projects/superfoo/branches
svn mkdir svn+ssh://vision.cs.arizona.edu/home/svn/src/projects/superfoo/tags
```

You then move your constructed directory to a safe place, and checkout the copy you just put into svn.

```
cd ..
mv foo just-in-case-backup-foo
svn co svn+ssh://vision.cs.arizona.edu/home/svn/src/projects/superfoo/trunk foo
# verify that the new foo looks fine
rm -rf just-in-case-backup-foo
```

Yes, this is a bit confusing and awkward, but you don't have to do it very often.

4.5.12 Reorganizing files

If you want to reorganize a project, you'll probably want to rename files, delete some, maybe make copies, and move around directories.

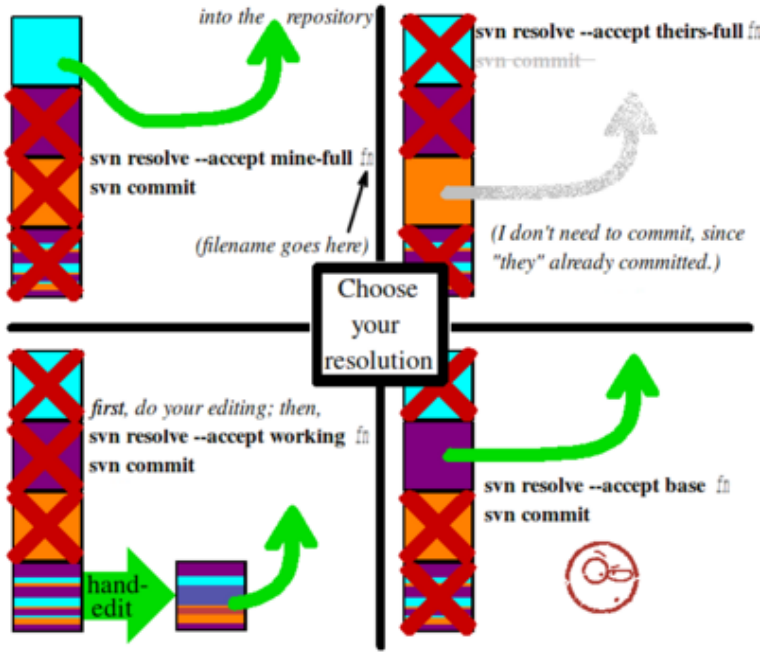

Figure 4.5: Four different ways to resolve a conflict (choose one)

Subversion has analogues of the basic Unix commands: "`svn cp`," "`svn mv`," "`svn rm`", and "`svn mkdir`". There is no "`svn rmdir`" – use "`svn rm`" instead. Whenever you want to modify a Subversion-controlled directory, and also have those modifications reflected in the repository, be sure to use the Subversion analogues. Remember that any local changes stay merely local until they are committed to the repository.

4.5.13 Version number subtleties

The Subversion `Id` keyword is handy way to insert a revision number and some other identifying information into your source code automatically. All you need to do is insert a special string – "`$Id$`" – into your source code, and "activate" it to tell Subversion to expand this string when you access the file in the repository. The full explanation on "Keyword Substitution" can be found `http://svnbook.red-bean.com/en/1.5/svn.advanced.props.special.keywords.html`. But if you'd like a quick explanation by means of example, read on.

Assume you wrote a c++ program name `hello_world.cpp` and you want to use the `Id` keyword in it. You can just add the following lines at the beginning of your cpp file:

```
/* $Id$ */
```

or

```
/*
 * $Id$
 */
```

Subversion won't expand these sorts of strings unless it has been told to do so: you must "activate" the keyword to make it work. To do so, you can use the following command

```
svn ps svn:keywords Id hello_world.cpp
```

`ps` stands for `propset` Once you have done so and commit the file, the line was expanded by Subversion like so:

```
/*
 * $Id: hello_world.cpp 3714 2009-04-10 22:06:17Z USERNAME $
 */
```

As you can see, now it shows the filename, the revision number of the last commit (3714), and the date, time, and programmer (USERNAME) who made the last commit. The letter Z next to the time indicates that it is UTC.

Unless you take further steps, you would have to use the "svn propset" command to activate the `Id` keyword in each file you want the keyword string expanded. (You can use `svn propset` with multiple filenames to make this easier.) If that seems tedious and onerous to you, then you might want to know how to activate these keywords implicitly. To tell Subversion always to expand these strings, you edit your `~/.subversion/config` file in sections `[miscellany]` and `[auto-props]`. Details are omitted, but a blog entry explaining this is found at http://makefile-it.blogspot.com/2008/11/subversion-and-tag-id.html or http://www.startupcto.com/server-tech/subversion/setting-the-id-tag

4.5.14 Fancy SVN tricks and tips

Copy from the wiki pages — lots of useful stuff.

Here are a set of useful SVN command:

```
svn st
svn diff
svn log
svn blame
svn relocate
```

To check how to use them, use "`svn help [COMMAND]`" and replace "`[COMMAND]`" with `st`, `diff`, and etc.

"What have I done?" (status, diff)

If you've worked with a checked-out copy and want to know what changes you have done, you can find out. The "`status`" command will summarize the status of changes in a simple binary sense (which files have changed and which haven't). The "`diff`" command will give a detailed, line by line display of modifications.

Type in "`svn status`" or "`svn st`" for a brief summary of status next to filenames. The status codes are a little cryptic at first, so you might also want to try "`svn help status`." By default the status command does not reveal whether the repository has changed, but you can see that information with the "`svn status -u`" version of the command.

Line by line changes can be examined with "`svn diff (filename)`" or "`svn diff`"; you might want to pipe this output into `less(1)`. If you have set up a tab size of 4 spaces in vim (see section on vim?), you can always do this to make the diffs look right:

```
svn diff | less -x4
```

Rules of Thumb

- Update before commit
- Follow lab conventions for directory structure
- Commit your files often!
  - Usual rule: commit as soon as it compiles OK
  - Don't wait until it is fully tested
  - Don't let too many hours of work accumulate between commits
- Use the `$Id$` string
- Avoid symlinks in working copy directories
- Branching is a big deal (check with all stakeholders)
- What else?

Fancy Diff using Vimdiff

You might need to go back to this section after you learned vim editor

(Based on this blog.) Plain old diff output is usually adequate, but sometimes you really want to make a side-by-side comparison (instead of interleaving the changed lines). In these cases, vimdiff is a fantastic tool for comparing two files. If you would like to use vimdiff when you do svn diff, there's a way:

1. You need to create a script like the one below, and
2. You need to point Subversion to use that script by editing your `$HOME/.subversion/config` file

The script you need can be as simple as the following:

```
#!/bin/sh
vimdiff $6 $7
```

If you call that script "diffwrap.sh" then just edit your `OME$H/.subversion/config` file like so:

```
. . .
[helpers]
diff-cmd = /path/to/script/diffwrap.sh
. . .
```

For more information on vimdiff, see section on vim.

want to Tag! I want to Branch! I want to Merge! Advanced topics, eh? We've covered the basics (and a few words on tagging http://vision.cs.arizona.edu/wiki/Two_Minute_Explanation_of_Tagging)(*do we need to move the details of tagging here?*); for the rest, we recommend the Subversion book (http://svnbook.red-bean.com/).

Relocating

Sometimes the location of the repository will change, and your working copy won't be able to access it anymore. Sometimes this is because your working copy is on a networked drive and you're accessing it from a different computer. Other times, the actual location of the repository changes. There is a quick and easy way to update your working copy to point to the new repository location. From the root directory of your working copy, use the `switch` command with the `--relocate switch`:

```
svn switch --relocate <<old location>> <<new location>> .
```

Note the "." at the end of the command, indicating that the current directory should be updated (as well as all child directories). If you don't know the current location of the repository to use for «old location», just type:

```
svn info
```

Trac The KJB Library has a graphical interface for browsing at http://vision.cs.arizona.edu/trac/kjb. The username is `wiki`, and the password is `wiki4fun`. To browse the source, click on the 'Browse Source' button in the navigation bar at the top of the page. This interface can also be useful for comparing code from different branches and revisions (so don't be afraid to delete old code instead of commenting it out!)

4.5.15 To git or not to git, that is the question

XXX

4.6 Command line environment

KOBUS SAYS: Very much under construction. ←

KOBUS SAYS: There is a bit of synchronization needed with § 4.1.2 ←

While the whole point of setting up your environment is to have it tailored to your way of thinking, some of the ideas, scripts, and config files written by others may be helpful, either as is, or a starting point. Some sources are listed below (SOON).

However, it is inevitable that you will need to subscribe to:

```
svn+ssh://vision.cs.arizona.edu/home/svn/src/Make/trunk
```

which you could do, for example, by:

```
cd
mkdir -p src
cd src
svn co svn+ssh://vision.cs.arizona.edu/home/svn/src/Make/trunk Make
```

Further, it is inevitable that you will need to have the `scripts` subdirectory, e.g., `src/Make/scripts` if you installed it as above, in your shell `PATH` variable.

Chapter 5

IVILAB software system (IVISS)

This chapter is about the IVILAB build systems, development protocols, and the existing library code base (libIVI). The string "IVI" and "ivi" indicates components within, and avoids name clashes. Note that this library was previously called "KJB". Until "KJB" is purged from code bases and our memories, the strings "KJB" and "IVI" (and "kjb" and "ivi") can be considered to be the same.

5.1 Prerequisites

The IVI library depends on the presence of several core *nix programs and libraries, primarily for building software and numeric computation. Additional libraries are optional for the core, but might be needed by specific programs and/or if available can significantly improve performance.

5.1.1 Required programs

- Basic utilities that init_scripts will insist it can find. Most of these should be on your system by default except the ones in bold font discussed further below. You can test their using "which", which takes multiple arguments so you can do the whole list in one go. This list should be in synch with the list in init_scripts.

  test cat cp mv rm mkdir file grep sed ls ln touch chmod gzip **makedepend** date tee egrep tr

- Basic utilities for some tasks that init_scripts will warn about not being able to find (this list should be in synch with the list in init_scripts):

  awk egrep stat tail

- clang (mac) or gcc, g++ (unix), make, etc
  Ubuntu: build-essential
  Mac: install xcode command-line tools from the command-line:

```
$ xcode-select --install
```

- tcsh
  Already available on Mac and Ubuntu, but sometimes not on other systems.

- vim
  Usually there by default.
  Homebrew: macvim

- makedepend (comes with ubuntu's xutils-dev package)
  MacPorts: makedepend
  Homebrew: makedepend
  Most of what makedepend does for us can now also be done by recent GNU based compilers, which includes clang. We may switch to that soon.

- ctags (for tags)
  Usually available on Mac and unix, BUT the default mac one is not the right one for us. Install the correct on using mac ports. Test that you have the right one by typing "ctags –version". We use "Exuberant Ctags". The default mac one does not even tell you what version it is (i.e., the just mentioned test command fails with error status).

- doxygen (for C++ documentation)
  MacPorts: doxygen

- bison or yacc (for C documentation)
  One or the other will do. Usually installed on linux systems.
  MacPorts: bison

5.1.2 Required libraries

The following are listed under their Ubuntu package names, grouped by functionality, with some notes about Mac installation using Homebrew or MacPorts. If the exact Ubuntu package name is not available for your platform, you can search for the comparable package starting with these names.

- libncurses, libncurses-dev
  Mac: Already should be there.

- libboost : Used by most C++ code. Not relevant for C code.
  MacPorts: boost
  Homebrew: boost@1.71 (example for version 1.71)

  The boost version has plagued libIVI for years, and we welcome help on our effort to make the system and the programs it supports robust with respect to boost. Recent (summer 2020) testing suggests that the core system can be built with boosts in the range of 1_69 and 1_71. If your boost is outside this, you might run into trouble (but try it and report). You can force the version of boost using the environment variable IVI_BOOST_VERSION, either in the shell, or in the file Make/BUILD_ENV, which has an example of the syntax, or in files listed in the file Make/BUILD_ENV.example. Ideally, we will make very little of these "stop gap" facilities, and fix the code to be more flexible with boost versions, or at least robust if the version of boost is late enough.

5.1.3 Optional libraries

KOBUS SAYS: This could use some cleanup. Having dev versions and non-dev versions in these lists is confusing. Usually dev implies non-dev on linux. Is this supposed to be mac vs linux? ←

These libraries are optional but recommended as they are used by a lot of the code base. However, we protect against missing optional libraries as described below (XXX).

- libgsl
  MacPorts: gsl
  Homebrew: gsl

- liblapack-3gf, liblapack-dev, gfortran, libgfortran3
  MacPorts: gcc46
  Homebrew: gfortran

- libx11, libx11-dev
  On Mac, install X11 tools from install cd)

- libjpeg, libjpeg-dev, libpng, libpng-dev, libtiff, libtiff-dev
  MacPorts: jpeg, tiff
  Homebrew: ?

- libgl, libgl-dev, libglu, libglu-dev, libglut, libglut-dev

- libsvm, libsvm-dev

- libcv, libcv-dev, libcvaux, libcvaux-dev

- libglew
  MacPorts: glew
  Homebrow: glew

KOBUS SAYS: Which programs link against this? We have a hacked ⟵ version in the library

- imagemagick libmagick-dev
  MacPorts: imagemagick
  Homebrew: imagemagick –with-magick-plus-plus

There are more libraries that are used with specific IVI lab programs, but these are the basic ones.

5.1.4 Checking out the IVI build system and library

Now that you've installed the prerequisites, we'll set up the IVI build environment. This has been done for a long time using SVN, but you can use GIT as described below.

SVN oriented instruction

First of all, we need to create a directory where everything will live (usually named 'src').

Inside src, we must check out a number of directories from the SVN repository: lib, Make, include_after, include_before, and any other templates we need (see below).

You must have enabled tunneling into the CS network and Passwordless SSH keys if you are installing the files onto your own private laptop – see Section 4.4. (That means you can type "ssh vision" and immediately log into the vision machine at vision.cs.arizona.edu; vision is used here, you might need to use the full name "vision.cs.arizona.edu" if you have not set up the alias.) In this case, you can check out the important directories like so:

```
svn co svn+ssh://vision/home/svn/src/Make/trunk Make
svn co svn+ssh://vision/home/svn/src/include_after/trunk include_after
svn co svn+ssh://vision/home/svn/src/include_before/trunk include_before svn co
svn+ssh://vision/home/svn/src/lib/trunk lib
```

The above is the minimal checkout. It is in fact easier to check out the core of the IVI software system by checking out the `Make` directory, and then using the script `update_ivi_core` to check out (or update) the core system. The definition of the core may change, but we expect that it will always include the four directories above. The incantation for this approach is:

```
svn co svn+ssh://vision/home/svn/src/Make/trunk Make
Make/scripts/update_ivi_core
```

This script is a good way to update the core system, including the Make directory. This means the update script will update itself also.

Cool folks will use GIT! [UNDER CONSTRUCTION — DONT USE YET]

You can get the above four directories (as well as some other directories you can ignore to start) by:

```
git clone  https://github.com/ivilab/src
```

The intention is that these two sources will be tightly synchronized using automated processes. As of July 11, 2020, the scripts are more or less working, and are about to be stress tested. Stay tuned!

Using the IVI library in multiple projects.

The standard view of a library module is that it is external to your code, and you import updated versions as they are attractive. However, the IVILAB development philosophy is that you build the library while you build your your program. That means that you need to have the source code for the library nearby your program(s), so that you solve problems in a modular way, for the collective benefit. This creates hassles with GIT which its concept of "entire" project. This means that either the project should be all IVILAB programs ever written, or we need a copy of the library. The situation is not perfect using SVN, except that SVN nicely handles subscribing to parts of such an "uber" project, which is how we have done things for a long time.

There is no perfect solution with GIT. The current plan is to have multiple copies of the library in several big projects that need them. As a programmer, you likely are uncomfortable with two pieces of code that should be the same. The solution we will try to is to tightly synchronize the copies using scripts. (As of June 12, 2020, the scripts are more or less working, and we will be testing them during the rest of the summer). Pull requests for projects involving library components should be done frequently, and other steps that reduce conflicts should be practiced.

5.2 The IVISS directory structure

The following is consistent with the layout in SVN, but only a subset of what follows needs to be that way. For example, the top level need not be called "src", and program directories can be anywhere in the tree. The tree is defined by all directories below the parent of the directory "Make". The directories `lib`, `include_before`, and `include_after`, should be siblings of `Make`. Typically `IVI` and `IVI_cpp` will also be siblings of `Make`, but they are not essential (except to build C++ documentation) and could be elsewhere in the tree.

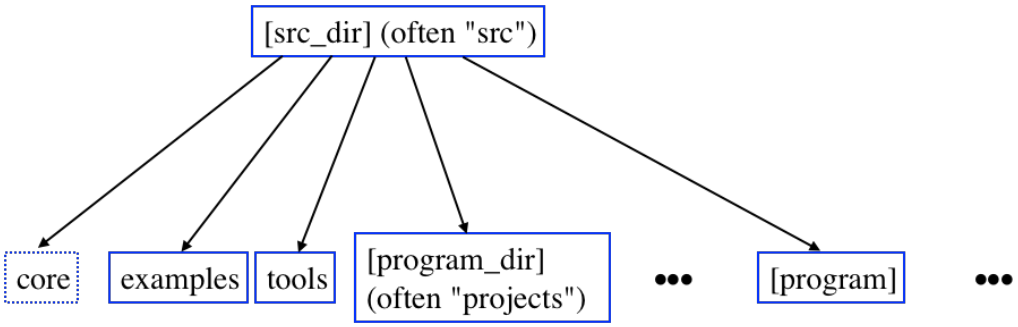

Figure 5.1: High level IVISS directory structure.

5.3 The IVI Build System

We use ~~a Procrustean~~ an artisanal system to build our software[1]. The IVI build system automatically figures out dependencies between your code and any IVI or 3rd-party libraries and creates makefiles accordingly. In addition, the system figures out if the makefiles themselves need to be updated, which, if it is available in other systems, it would be a recent development. Finally, the system works well for multiple builds on the same file system, ranging from development versus production, to versions of OS's, to versions of available components such as boost. Much of the capability of the build system was developed at a time when options were limited. For example, the system predates CMake. However, one can ask whether other systems are now up to the task of replacing our current system. There are several points that we need to be carefull about:

- The philosophy of CMake is to prove a powerful system to describe building your system. We instead provide a system that encodes what we have learned over the years to avoid specifying these things, and when IVALAB member A adds something to deal with library X, other folks using the library X do not need to figure it out.
- Some of the features we like, such as knowing when makefiles need to be rebuilt, or seamless multiple concurrent builds, are not necessarily available, or are not implemented as we would like.
- The build system, while far from perfect, is there for us to inspect and improve. We can add the features we want, and fix the issues that annoy us.

The scripts that make it up are in the Make directory of the source tree, and it is a good idea to be at least a little bit familiar with the most important ones. More information may be found here: programming/makefiles.txt

5.3.1 IVI `make` options

TODO: Provide summary explanation of what each of these does

```
make confess
```

[1] The preceding sentence was not constructed by Kobus, but Kobus will take responsibility for most of the philosophical points that follow

... lists dependencies (locations of headers) that the ivi build system can find.

```
make
```

```
make doc
```

```
make tags
```

```
make regress
```

```
make test
```

```
make ivi_code
```

```
make ivi_doc
```

```
make ivi_tags
```

```
make ivi_regress
```

```
make ivi
```

```
make clean
```

```
make cache_clean
```

```
make locks_clean
```

```
make depend_very_clean
```

5.3.2 Manipulating the build

TODO: Some general stuff, including what bash users do.

Compiler flavors and versions

TODO

Boost (and other library) versions

TODO (some overlap with the boost stuff above. Perhaps move that down here and have a pointer to here).

Development and Production Code

KOBUS SAYS: THERE IS SOME OVERLAP HERE WITH THE BUILD SYSTEM. INTEGRATE! ←

The IVI library can be compiled as development code, or as production code. If you are using Kobus's setup, then the aliases

```
set_dev
```

and

```
set_prod
```

can be used to switch between them. If you are not using this environment, then you need to set the environment variable PRODUCTION to 0 or 1 before building.

The general method is to put the objects of the various ways to build code into subdirectories of machine depending directories. This means that the path to the program that you just built is a bit obscure. If you are using Kobus's setup, then when you switch the kind of executable, then the PATH variable is adjusted so that the path to what is being build current is the first item. If you are not using Kobus's setup, this is detected by the build scripts, and a symbolic link is made from the last built versions of programs to the source directory. This behavior can be changed if needed.

KOBUS SAYS: The following needs to be improved ←

```
Other, more obscure, compiling methods:
   With           With tcsh                  What it does
   Kobus's        (less tested)
   setup

   set_no_libs    setenv NO_LIBS 1           Tests compiling without
                                             optional libraries
                                             (e.g. lapack).

   set_g++        setenv IVI_CC g++          Compiles C code with g++.
                                             (C++ is always compiled with a
                                             C++ compiler).

   set_cxx        setenv IVI_PREFER_CXX 1    Try to compile C code with a C++
                                             compiler (not necessarily g++).

   set_debug      setenv FORCE_DEBUG 1       Compiles with debug flag. Use
                                             this if you have to debug
                                             optimized code. Use "dev" for
                                             more standard debugging.

   set_no_opt     setenv FORCE_NO_OPTIMIZE 1 Compiles without development
                  setenv FORCE_OPTIMIZE 0      features but without optimization.
                                               This is useful if you run into an
                                               optimizer bug which happens on
                                               occasion. Having concluded this,
                                               there is a way to force various
                                               levels of optimizations in the
                                               Makefiles.
```

Development code is normally compiled with the flags "-DTEST -g". Production code is normally compiled with some optimization, and no"-g" flag. In addition to the above, development code provides other useful facilities

described below. With a few exceptions, these facilities are not available with production code.

5.3.3 IVI scripts

KOBUS SAYS: I started on Clays TODO about getting scripts documented. ←

Many of the build scripts that live in `$SRC_DIR/Make/scripts` are documented in Appendix §B. Some that you are more likely to use are introduced next.

```
Make/scripts/make_ivi
```

... crawls ivi code to do various tasks ranging from checking that it all compiles under various conditions to running tests to building standalone versions for export. Some of the more basic operations are available in the build system, which calls this script with appropriate parameters. E.g., to compile the core library in several ways:

```
make ivi_code
```

See also make targets `ivi_doc`, `ivi_tags`, and `ivi_regress`.

5.4 The IVI Library

The IVI library is the vision group's software library. It has both C and C++ parts to it, and it is where most of our code comes from and ends up. It is mostly used for mathematics code (like linear algebra and statistics), but spans other areas, such as image processing and machine learning. It is made up of units called "sub-libraries", each of which specializes in a different area of functionality. Along with the code, we have several coding standards we like to adhere to, for things such as error handling and memory management. Anybody that codes in our lab should be very familiar with the IVI library. The IVI library attempts to adhere to certain programming conventions, which we are gradually documenting.

5.4.1 Accessing IVI man pages on an ivilab server

If you're using the IVI library, you'll probably need to use the library man-pages. To see if they are available:

```
man ivi
```

If nothing is found, the following path needs to be added to your shell profile:

```
/work/vision/doc/man
```

In csh/tcsh Shell

If you are using csh/tcsh, edit your .cshrc:

```
vim ~/.cshrc
```

In bash Shell

If you are using `bash`, edit the `.bashrc` file.

```
vim ~/.bashrc
```

Look for a line in the file that says

```
setenv MANPATH ...
```

and add `/work/vision/doc/man:` to the beginning of the manpath string (note the trailing colon ":"). If the MANPATH line doesn't already exist, add it at the end of the file.

Finishing up

When these paths have been added, log out and log back in, and you should be able to access the man pages. To test it:

```
man ivi
```

5.4.2 Generating Man Pages

Sometimes you'll notice a man page is out of date, so you'll want to generate a fresh version. Or, perhaps you would like to build them on your laptop (good idea!). Generated man pages are saved in `/work/$USER/doc/man`, but we want them to be saved in `/work/vision/doc/man`, so everyone can benefit from the updated man pages. In order for this to occur, set up a symbolic link as follows:

```
ln -s /work/vision/doc/man /work/$USER/doc/man
```

In this example, we'll assume you want to update the documentation for the `gauss_rand()` function, which is defined in `lib/sample/sample\_gauss.c`. To generate the new man pages, enter directory containing the source code for the function with stale documentation, and run "make doc":

```
cd lib/sample
make doc
```

Now check the documentation for `gauss_rand()`

```
man gauss_rand
```

The man page should reflect the updates.

5.4.3 HTML version of the man pages

The HTML version of the man pages can be found here:
http://kobus.ca/research/resources/doc/ivi/ivi.html

The Doxygen pages for the IVI C++ Library can be found here:
http://kobus.ca/research/resources/doc/doxygen/index.html

5.4.4 Generating tags

The build system can build tag files for all identifiers relevant to the code in a given directory. We go the extra mile to provide tags for system includes, which is useful for tracking down boost issues. However, system includes often have fancy macro definitions that confuse ctags. We can (and should) patch these via the file Make/ctags/identifier_hacks.txt.

To build tags in a given directory use "make tags". To build tags in all core directories, use "make ivi_tags". Currently we build two tag files

"tags" and "tags_h". The first comes from C nd C++ code, the second from header files. One reason for not combining them is in the hope that vim will eventually provide more options for how multiple tag files are consulted. Currently, there is little difference. However, unless you are not interested in the header file tags, you will want to use the following line in your .vimrc file:

```
set tags=./tags,tags,./tags_h,tags_h
```

5.4.5 Trac Web Interface

KOBUS SAYS: OUT of DATE: Either update or declare no longer needed, perhaps due to upcoming integration with git. ←

The Trac web interface for the IVI Libraries can be found here:
http://vision.cs.arizona.edu/trac/ivi

ADARSH: Copied from wiki

5.5 Adding external libraries

To tell the IVISS about an external library, we edit the file init_compile. We have vague plans to modularize this file, but for now all external library information is in this file. The instructions to do so are at the beginning of the file.

5.6 Using IVI C/C++ from Python

We are currently experimenting with two ways to call IVI C/C++ code from Python. The first uses Python's `cdll` and `ctypes`, and the second uses Pybind11. We will refer to the first method as ctypes and the second as PB11. The directory `src/examples/python_interface` bas examples of each, and further instructions in README.txt files.

The ctypes method is simpler, and might be viable for functions that use simple data types. It is designed with C in mind, but can be used with C++

if you declare your function to have have C bindings. The PB11 is more comprehensive and has extensive support for C++.

In both cases, you need to write C/C++ wrapper code that implements the functionality you need. Often this would simply be a lightweight wrapper for some IVI C/C++ function. The wrapper code is a bit different for the two cases (see the examples). Having created the wrapper, you use `make pylib` (for ctypes) or `make pb11` (for PB11) to create a shared object (.so file) that can be loaded with `cdll.LoadLibrary()` (ctypes) or imported with `import` (PB11), typically dropping the suffix. Likely you will want to do the `make` in PRODUCTION_MODE.

5.6.1 Pybind11 pitfalls

Pybind11 is powerful, but there are pitfalls. Currently, it is recommended that users of Pybind11 use link time optimization (LTO), but we are leery of that based on needing to back off, and currently do not use it. In addition, one needs to be careful about conventions about copy constructors versus Python's call by reference.

KOBUS SAYS: Manujinda can add more information here.. ←

In general, one should check that all the data passed between C++ and Python be checked as the first step in implementing the functionality.

If you are developing with pybind11, most things will work 'out of the box', however, there might be times where you might have to deal with the complexities of integrating Python and C++. Some subtle issues arise from the different ways in which Python and C++ manage memory - so you might want to keep the contents of this page in mind: `https://pybind11.readthedocs.io/en/stable/advanced/functions.html`

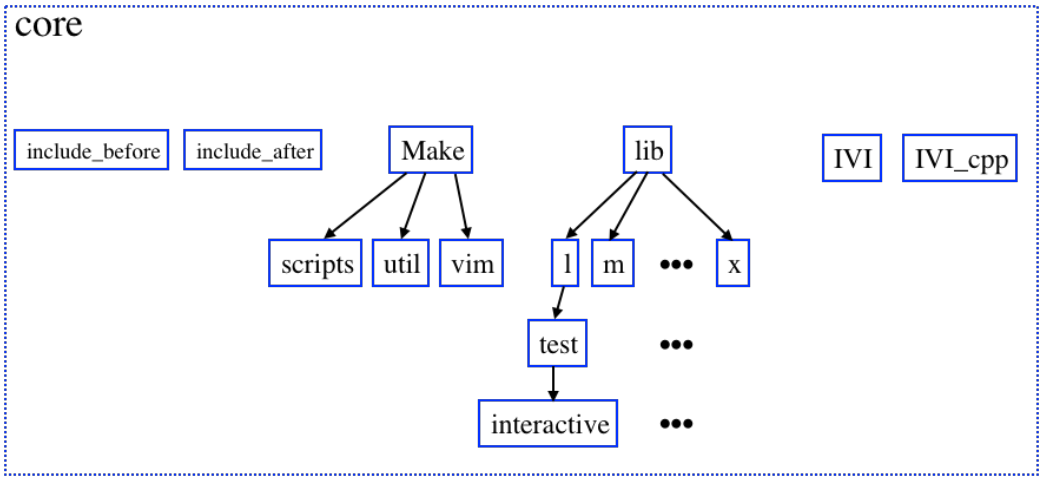

Figure 5.2: Details of the core IVISS directory structure.

Chapter 6

Software Development

6.1 IVILAB C/C++ Coding conventions

Adapted from material written by Kobus alone. Hence "I" means Kobus.

For code which is to be worked on by several people (especially when those people include Kobus — e.g. library code and projects extending the existing code base), the following conventions hold. They are also strong suggested as a starting point for those with no strong opinions, and those with strong opinions are urged to discuss them with others. There are several protocols which IVILAB folk picked up after a number of years of coding — i.e., there exist reasons for them that may not be immediately apparent.

6.1.1 Language versions

Generally, the lower level the code, the more compilers it should be compatible with. C library code, especially the "l" and "m" sub libraries are best kept at strict ANSI C (c89). However, we currently build with strict c99, which means, among other things, that "//" can be used for comment lines, which is fine for non-library C code. C code should also compile with a C++ compiler.

Currently, we do not have a rule about which C++ standard we should conform to. Until we get consensus on this, we should err on the side of simplicity. Again, library code, should be kept at a low common denominator.

In general, if you use a recent fancy feature, then your code may not compile on someone else's system. C++ is a very complex language. There is no requirement to use every feature!

6.1.2 Compiler warnings

We generally ask for most warnings, and we strive to keep the number of warnings small. By default, the IVI makefiles ask for a reasonable set. Don't ignore warnings you don't understand. We want to keep the number of spurious warning to a minimum so that valid warnings are easier to notice. There is some facility for filtering out warnings that can be ignored in the current context, and likely more will be added. For example of an existing filter, by default, the IVI build system strips out certain warnings that the C++ mostly header library "boost" produces, as we cannot do anything about them, but we want to ask the compiler for those same warnings in case they come from our own code.

6.1.3 Formatting

Arguably, formatting cannot be very important, since there are tools that can change the formatting to any style you like. In particular, VIM can format your code as you type, and reformat it later. However, if the files in a code base are formatted similarly, then the code is easier to read. The following formatting conventions are generally followed in the code base, and code that does not follow it will eventually be reformatted by the code police.

If you are using Kobus's setup and vim, then the formatting rules can be inherited by copying `~kobus/.vimrc.ivilab`. This file does the syntax coloring as well, which tends to look best with a light background, but it is easily tweaked if you want to use it.

White space. Each line of source should generally be no more than 80 characters wide (soft convention). Use blank lines generously to break up the source into meaningful chunks. We use 4 space indents. We prefer spaces to tabs. If you use tabs, then many things, including the amount of indent is a function of your editor! To convert a tabbed file to one with spaces in vim, use "retab".

I prefer spaces after commas as in: `foo(a, b, c, d)` instead of `foo(a,b,c,d)`. The same often applies to semi-colons: `for(i=0; i<3; i++)` instead of `for(i=0;i<3;i++)`.

Braces: The biggest issue We put the { and the } in matching columns. Thus we use:

```
if (a)                          if (a) {
{                  NOT
}                               }
```

We prefer using {} even when there is only one statement inside. It is easier to read, and you will invariably want to add something anyway. Also, if what you have is a macro or compound statement, you may have a bug! So, we prefer

```
if (a)                                  if (a)
{                                            one_thing();
   one_thing();             NOT
}
```

Exceptions: Sometimes it is fine, or even better to do it the other way, but typically we put everything on the line in that case. For example.

```
if (a) break;
```

The same can apply to while() and for().

If all the parameters for a function do not fit on one line (often because of comments), then we use library formats the function like so:

```
static int get_blob_word_posterior_matrices
(
    Matrix_vector_vector** trans_mvvpp,   /* Stuff about this one. */
    Word_list**            labels_pp,
    const char*            dir,
    int                    data_set
)
{

/* Code goes here. */

}
```

6.1.4 Formatting function block comments (C)

UNDER CONSTRUCTION

6.1.5 Formatting function block comments (C++)

UNDER CONSTRUCTION

6.1.6 Scope and abstraction

Scope. We limit the scope of any entity as much as possible. This means that ONLY things that are exported to other functions go into .h files. If a function is needed only in one file, then it should be declared static and only put in that file. The same applies to #define's, typedef's, and global variables which should be file statics if at all possible. A .h file is NOT a general list of declarations. It is a list of declarations which you want to export. Limiting scope also means that functions and global variables not exported should be declared static.

Abstraction. You should not assume anything about the details of data structures and functions beyond what is documented. If this convention is followed, that means that they can be improved later without creating trouble.

6.1.7 Naming guidelines and conventions

It is important to use good names. You should edit your code to improve it, even after it works, and names so that the code is more readable to others and even yourself (in a few days) is a bit part of what you want to check. Other things include flow and logic — is it obvious that this code will work, or at least fail a check if it does not?

Many of our naming conventions express semantics. Some also make it so that semantic free syntax coloring works in the "vim" editor. We prefer complete and expressive names except when a short name is clearly appropriate (such as "i" for a counter). I will tend to spell out a full name rather than abbreviate, until doing so becomes too unwieldy. There are a few exceptions where I have found the convenience and terseness gained is sufficiently great. One example that comes to mind is the function "pso".

On average, names with large scope tend to be longer, or at least protected by some prefix, but there are many exceptions to this. In the case of the IVI library, the prefix used to reduce naming conflicts is "kjb_". Not all routines begin with this prefix, just the ones deemed likely candidates for conflicts. (There are some inconsistencies here which maybe should get fixed).

Case. For variable and function names I prefer lower case and underscores. In other words, I much prefer invert_matrix to InvertMatrix. I find the later hard to read and unaesthetic. On occasion, the use of upper-case in a name can be more expressive, and thus there are exceptions. Finally, you should never use an underscore as the first character of one of your identifiers, as these can have special meaning.

Functions are actions, variables are things. For C code, at least, I suggest making function names sound like actions. Thus I prefer copy_matrix to matrix_copy. Variable names on the other hand usually sound like things. Thus the reader of "thing = get_thing_from_stack()" does not have to get to the "()" to know that the RHS is a function and not another variable. I find this helps make the code more readable.

In C++, "everything is an object"[1], but we should establish some naming conventions for C++ at some point (TODO).

Macros. Macros are normally all in upper-case. I also often make enum's upper-case which breaks the more common convention of having enum's in lower-case. The logical role of enum is more like a #define, and less like a variable, so the code is easier for me to read because I equate upper-case entities without parens as constants.

Action macros which may conflict with others are "fixed" by adding "_OF" For example, I use MAX_OF() instead of MAX(), which everyone else uses.

Types. User defined types (via typedef) follow the same style as variable names, EXCEPT that the first letter is upper-case. Thus "Matrix" is a type, whereas "matrix" is a variable. I use this convention for syntax coloring, as well as solving the common problem of wanting the type and the variable the same name in a consistent manner.

Some function name conventions (C). Functions do things, and often a good name is suggestive of action. For example, "read_file()" is better than "file_read()"

Functions is_<x> return either TRUE or FALSE (and possibly ERROR which is specifically defined so that it is not TRUE, FALSE, or EOF). Such routines generally test whether the argument is X or the state X is the case.

Functions check_<> return either NO_ERROR or ERROR or call exit. They generally check some condition which may indicate error. If they return ERROR, then the appropriate error message has been set. For example, the code to subtract two image has, as a first line:

[1] I really do not know what this often cited platitude actually means.

```
ERE(check_same_size_float_image(in1_ip, in2_ip));
```

The prefix ow_ indicates that the first argument is over written. This is normally used in the case that a similar routine without the over write exists, or will likely be written at some point. The non over-write routines are more standard in the IVI library and more versatile, but sometimes the over write ones are more useful and more common. For example there are two routines to scale a matrix :

```
multiply_matrix_by_scalar(target_mp_ptr, source_mp, scalar)
```

and

```
ow_multiply_matrix_by_scalar(source_mp, scalar)
```

The first one does not touch source_mp, and the result goes into *target_mp_ptr, which is created if necessary. The second one overwrites the matrix source_mp with the scaled version. Both routines are useful, and the prefix helps distinguish the two behaviors consistently.

A function with the prefix "fp_" works on a file pointer in the case where there is a similar routine (or a proposed similar routine) which works on a file name instead. In this case, the routine taking the file name as a parameter will be implemented using the "fp_" version, with the convention that a NULL or " filename indicates stdin or stdout. For example, we have read_matrix() and fp_read_matrix().

A function with the prefix "get_" normally returns ERROR/NO_ERROR, putting the result into a location provided by a parameter which is a pointer, whereas the same function name if it exists will return the computed result directly. For example:

```
mag = vector_magnitude(vp);
result = get_vector_magnitude(vp, &mag);
```

The routines for creating (in C) "objects" have the prefix "get_target_" They have a double pointer for the first argument, and return ERROR/NO_ERROR. In general, a function of the type "get_target_" creates an object if it does not exist, resizes it if it does exist but is the wrong size, and does nothing if the size is correct.

For common objects, there usually is a initialized version (e.g., get_initialized_matrix(), and even specific initialized versions (e.g, get_zero_matrix(), get_identity_matrix(),

get_unity_matrix(), and get_random_matrix()). Notice that these routines typically have the word "target" replaced with something more specific.

Largely due to historical reasons, there are also routines which create objects and return pointers in the more standard way. These have the prefix "create_". In general, this style should be considered OBSOLETE as part of library code as there is a real gain in consistency, flexibility, and efficiency to use the get_target_xxx() style. The main current use of create_xxx() routines is as static routines which are used by the get_target_xxx() routines. At some point all "create_" routines that are not statics already will become statics.

Some variable name conventions. I tend to encode the number of times a variable needs to get de-referenced in the name. However, the extreme of this practice – i.e. Hungarian notation – really sucks. In addition, I will often break this convention when it seems that the code may be more clear without. Nonetheless, I use some a few simple conventions to indicate variable type which is especially useful for code examples, and function prototypes (making counting stars less critical). The current conventions are as follows: To make things clearer, I will often add the suffix _ptr to a generic pointer. Also, there are a number of abbreviated cases:

```
_fp    pointer to FILE
_mp    pointer to Matrix
_vp    pointer to Vector
_mvp   pointer to Matrix_vector
_vvp   pointer to Vector_vector
_sp    pointer to Spectra
_ip    pointer to Float_image
```

A pointer to these pointers often gets an additional "p" So a pointer to a pointer to FILE, would likely have the suffix _fpp, and similarly for the others.

6.1.8 Function Parameter Conventions

Unfortunately the IVI library code is not completely consistent with respect to the order of parameters. This is partly due to trying to be consistent with external conventions which themselves are quite inconsistent. In short, one must consult the documentation or header files. (This could use a bit of an overhaul if we were prepared to have significant breakage!).

Generally when a single result is computed (i.e. this normally applies to a mathematically oriented function), a pointer to this object will be the first parameter. Normally the object being computed will be created if it does not already exist, and resized if necessary if it does. An example is :

```
add_matrices(target_mp_ptr, first_source_mp, second_source_mp)
```

The order of the parameters is an attempt to be consistent to what I believe is most common, and thus is what I (and I hope others) expect. I believe this is due to the resemblance of the above to :

```
target = first_source + second_source
```

This reasoning does not seem to apply to generic procedures, especially in the case of multiple input parameters, and thus I will tend to write input parameters followed by output ones:

```
function(input_1, input_2, result_1, result_2)
```

(Unfortunately, there is a some inconsistency here, which should be repaired).

6.1.9 Creation and destruction

KOBUS SAYS: There is overlap with the file double_ptr.txt. Check that we have got everything that is useful into here. ⟵

Except in the few cases where the purpose of the routine is to create or destroy, your routines should destroy what they create. Files that are opened should be closed. The general approach to memory management (see below) is to set a pointer to an object to NULL when declared, pass the address of that pointer (double pointer) to routines that create and resize as needed in the body of the routine, and then finish with a free routine. IE:

```
Matrix* c_mp = NULL;   /* Must initialize this to NULL !!! */

while (more_to_do && (result != ERROR))
```

```
    {
       /* stuff */
       result = add_matrices(&c_mp, a_mp, b_mp);
    }

  free_matrix(c_mp);
```

Note that:

- Since free_matrix() can take a NULL, we don't have to worry if the loop gets executed. It is always safe to pass a destruction routine in the IVI library a NULL pointer.
- The first time the loop gets executed, a matrix *c_mp is created.
- Subsequent times through the loop, the storage gets recycled in the common case that the current size is correct.
- If the size is incorrect, there is no problem. The matrix gets resized. (This is done efficiently).
- If the routine fails (e.g., add_matrices above) the following conventions in the library apply:
  - You cannot assume that the contents are in any sense valid.
  - You cannot assume that the contents have not been touched.
  - You cannot assume that it does not need to be freed.
  - You can, however, assume that it is safe to be freed.
  - You can also assume that an error message has been set. (See §XXX below).

6.1.10 IVI C memory allocation conventions

UNDER CONSTRUCTION

6.1.11 IVI C error conventions

It is very important to do a good job on error handling. This file discusses the conventions that help accomplish this without too much added effort.

The basic assumption that is made throughout the library code is that if an error has occurred, then a meaningful error message has been set. The normal way to set the message is with a call to the routine set_error(). The error may be printed using kjb_print_error(), or copied into a character buffer with kjb_get_error(). (It is easy to change error reporting because all error reporting goes through a handful of routines).

In order for the basic assumption to hold, if a routine comes across an error condition, then it must set an error message, and return a value indicating error (almost always ERROR, sometimes NULL). It is important to test all relevant calls for error returns. Under our convention, whenever a routine returns error, it is safe to simply return error because a good error message has already been recorded.

Thus a common mode of error handling in my code is that a routine many levels down sets an error message and returns an error value. The next level returns an error value as a result, and so on, until the top read-execute-write loop which reports the error with a call to kjb_print_error(). (Something like a manual rewind of the stack done automatically in C++ when exceptions occur). This works fine for processing where there has not been any memory allocation. If memory has been allocated, then additional steps need to be taken to fight memory leaks (sometimes, when the chance of error is small, I skip this part, but I often end up going back and fixing it)

The "error message" is actually an array of error message lines. There are enough lines for any reasonable purpose, but if the limit was to be exceeded due to a bug, then the further messages are ignored. Each line is in the array of message lines is stored without a return, and thus including a "
n" at the end strings sent to set_error(), etc, will cause double spacing — not normally desired. The routine set_error() clears the list, and sets the first entry. Messages of more than one line can be built up using the routines add_error() which adds a line at the end, and insert_error() which adds a line at the beginning. Finally cat_error() appends a string to the current line. The need for both add_error() and insert() can be seen in the following example.

It is sometimes the case that a middle level routine "knows" something that should go into error message which is set by a low level routine. Suppose

the middle level routine looks after a database, and the low level routine opens a file. The low level routine knows why an open fails, and even knows the name of the file, but it does not know that this file is an index into the personal database. Often the best error message should contain both pieces of information. Thus the middle level routine can test for error on the open and add a line either before or after the message using insert_error() or add_error().

Additional fine tuning of the error handling is available through the routines set_error_action(), and get_error_action(), the later invariably being used to get the current value in order to restore it later. Routines are expected NOT to change the error action, and thus this save/restore step is normally needed.

This extra fine tuning is needed only occasionally. One instance is when we are continuing on, despite errors, and we want additional errors to be added to the current ones. Then, once there is an error, one could use: push_error_action(FORCE_ADD_ERROR_ON_ERROR); We have to wait for the first error, because that first error must over-write the (now obsolete) previous error. Once we are done adding errors, we then restore the error action so that the caller knows the error semantics with:

```
pop_error_action();
```

All the routines mentioned in this file have man pages, which should be consulted for the details on their use.

Tread safety. The error handling subsystem is currently not thread safe. Fixing this defect is not overly difficult, and will be done eventually. In the mean time, one can use push_error_action(FORCE_ADD_ERROR_ON_ERROR) or set_error_action(FORCE_ADD_ERROR_ON_ERROR) so that no messages are lost during threaded execution. The messages can, however, potentially be a bit mixed up.

6.1.12 Robust code development

KOBUS SAYS: THIS SECTION IS UNDER CONSTRUCTION. ←

Debugging Output

The IVI library has a number of I/O routines and macros which can be useful for debugging. In general, these routines can be used both in production and development code, but the amount of output (if any) produced by default can be different in the two cases. For example, the output of the "db*" macros is a function of the debug level. If the IVI options are being made available to the user, then this can be changed with the "debug"option. The defaults for development code is 2, which means that the output is printed using IVI library routines. In the case of development code, the default is 0, which means that the output is suppressed. Often our programs have a generic system for setting user controlled options, and normally the debug level is included int the options that the user can control. The debug level can be controlled in the code using the routine kjb_set_debug_level();

There are man pages for the following routines:

```
db_mat, dbm, dbo, dbp, dbx, dbc, dbf, dbi, dbr, dbw, dbe, dbs, dbnc,
test_pso, pdo, print_open_files, hex_print
```

ASSERT macros (man pages are available)

We strongly encourage the use of ASSERTs. These are statements which are validated during development runs. The are completely removed for production compiles. The library has a number of ASSERT macros which give relatively comprehensive messages and optional abort using kjb_abort() if the program is running interactively. The ASSERT macros, which are documented, include

```
ASSERT()
ASSERT_IS_NUMBER_DBL()        /* Checks for nan */
ASSERT_IS_FINITE_DBL()        /* Checks for inf */
ASSERT_IS_NON_NEGATIVE_DBL()
ASSERT_IS_PROB_DBL()          /* A number, and in [0,1] */
ASSERT_IS_EQUAL_DBL()         /* Within epsilon */
ASSERT_IS_NOT_EQUAL_DBL()     /* Within epsilon */
ASSERT_IS_GREATER_DBL()       /* Within epsilon */
ASSERT_IS_LESS_DBL()          /* Within epsilon */
ASSERT_IS_NOT_GREATER_DBL()   /* Within epsilon */
ASSERT_IS_NOT_LESS_DBL()      /* Within epsilon */
```

Verify routines (man pages are available)

Checks on standard data structures which are only active with TEST (i.e., development compiles:

```
verify_matrix
verify_matrix_vector
verify_non_negative_matrix_vector
verify_probability_matrix_vector
verify_probability_row_matrix_vector
verify_non_negative_matrix
verify_probability_matrix
verify_probability_row_matrix
```

Crashes, Aborts, ETC.

The IVI library assumes that there is no point in dumping core in the case of production code. First, since these executables are stripped, there is only minimal debugging information in core files. Second, the program may be used by a naive user who has no idea what a core file is, and does not appreciate having them created, as they can be quite large (this is obviously less of a concern in a research environment.) Therefore, by default, any circumstances which can lead to a core dump in development code has a different action in production code. In production code the action is some combination of:

- Creating a log file and printing a message asking the user to pass it along to the programmer.
- Printing a message that a serious or fatal error has occurred
- Exiting

The circumstances where the above applies include :

- Signals which normally cause core dumps such as SIGBUS, SIGSEGV, SIGABRT
- The execution of the "set_bug" routine (often through related macros).
- Calling kjb_abort(), either directly or through ASSERT().

In the case of development code the action in the first circumstances is to dump the core after some cleanup. The action in the second circumstances is to prompt for an abort (and if you enter a "y", then the action is to abort with a core dump after some cleanup. The action in the third case is to abort with a core dump after some cleanup.

The reason why cleanup is desirable before the core dump, is that the signals which cause the core dump are caught both by the program, and any of its children. Under the assumption that most of the action is normally in the parent, then we want to make sure that the core reflects exactly that process. On many systems, this is not guaranteed, so we get rid of the children first.

In addition to the above circumstances, the following will cause a prompt for core dumps in development code but will be completely unnoticed in in production code.

- Attempt to free memory which was not allocated.
- Attempt to free memory twice. Unfortunately, this is not distinguishable from freeing memory which was not allocated. However, if one is aware of the two possibilities, it is relatively easy to sort things out.
- Attempt to free memory which has been corrupted beyond the requested size. (For example, allocate 10 bytes, and write to the 11th byte.)
- Attempt to free a pointer which has been sent to the routine watch_for_free().

For more information on the above, man the following routines and macros :

```
set_bug
default_bug_handler
set_bug_handler
SET_BUFFER_OVERFLOW_BUG
SET_CANT_HAPPEN_BUG
SET_FORMAT_STRING_BUG
SET_CONTIGUOUS_ARRAY_BUG
SET_ARGUMENT_BUG
```

```
kjb_free
kjb_calloc
kjb_malloc
watch_for_free
memory_used

kjb_cleanup_for_abort
kjb_abort
ASSERT
```

Memory Leak Testing

The development version of the library checks memory allocation along the lines of several commercial and free products. It was written this way before these were readily available. Since it actually works reasonably well with the rest of the library, we continue to use the home grown version in conjunction with *valgrind*. There is an advantage to having a basic memory checker always running during development, rather than having to remember or choose to run valgrind at some random point. On the other hand, valgrind checks some things the built in version does not. We run valgrind in conjunction with testing scripts. Of course, running it at other times can be useful as well.

By default, when development code exits normally, all unfreed memory is listed by file name and line number of the allocation. Although the basic facility is to provide the location of a "kjb_malloc", some common routines which have implicit calls to kjb_malloc() have been modified so that the unfreed list will refer to their location, not the underlying free. For example, get_target_matrix() can create a matrix if *target_mp_ptr is NULL. If you do not dispose of the matrix, the unfreed list will point to the get_target_matrix() call, not the underlying kjb_malloc().

If the printing of the unfreed memory is a problem, or if it is making the code run too slow (it *is* expensive), then it can be disabled, either with the user option (if your programming is using the kjb option support):

```
set heap-checking=off
```

Or the system call:

```
EPETE(set_heap_options("heap-checking", "f"));
```

Note: This option can only be set off. Once off, it can't be set back on.

Untested Code

We use the macro UNTESTED_CODE() to flag execution paths that have not yet been followed. This makes it so that when the path gets executed (development code only) a message to the effect that untested code is being executed in a certain file at at a certain line number. The message warns me that the result should be checked in the context of what happened in the newly executed code. After the code has been thus tested, the macro invocation is removed. If you remove it from a local copy of the library, AND are confident that routine is trustworthy, let the maintainer of that routine know, and the UNTESTED_CODE() macro will be removed from the library. There is a similar macro, SUSPECT_CODE(), to flag code that you are not confident is correct, but that you do not have time to investigate.

6.1.13 Check-list

- Includes to library xxx are of the form xxx/xxx_*.h
- Check return values
- Don't return error without setting an error message. The calling routine must be able to to assume that routines that fail have set an error, AND, if they do not return failure (e.g. ERROR or NULL) that everything is OK.
- Don't put anything (especially defines) in a *.h file that does not need to be exported. If only the *.c file needs it, put it there.
- Functions conceptually local to a file should be made such using "static".
- If you need to #define of something likely to cause a name conflict, protect it with a prefix. EG, EPSILON –> FANCY_ALGORITHM_EPSILON
- Minimize compiler warnings. Compile both with "dev" "prod" with C and C++.
- Library code should not leak memory, except on very unusual circumstances. Typically the only one that we tolerate is due to an unusual ERROR return.

- No tabs. If you use tabs, they will eventually get changed to spaces by the formatting police.
- Naming conventions followed, especially for exported names?
- Formatting guidelines followed?

6.2 Testing and profiling

6.2.1 Testing concepts and conventions

We strongly encourage good testing of all code that matters, which typically includes most library functions, as this is where the real computation is ideally implemented, and others will likely want to use it at some point. Hence the library directories will typically have a subdirectory called 'test', where a number of programs constructed to exercise the functions in that library subcomponent live. Much of the description that follows is biased towards testing library functions (i.e., unit testing), but the test scripts described below (XXX) apply to any program or set of programs.

Good testing is generally automated where possible. Currently we have some automated unit tests and also quit a few that are not automated but should be, which can provide excellent warm-up projects for newcomers. These not-yet-automated tests are often found in directories `src/lib/*/test/interactive`. As the name suggests, these test must currently be run manually.

Our current testing protocol implements two processes. The first concerns establishing test programs that run without issues and that any output they produce is correct. The second (regression testing) checks that correctness is maintained as code is modified.

Robust code will run without crashing, pass internal checks, memory checks, and have the correct return value and output. Further, the test cases should collectively provide good coverage of what needs to be tested. These concepts are discussed in some detail next, as they are not quite as simple as they seem. We begin with return values (often called return codes or exit codes) as they are relevant to crashes and internal checks, which are discussed subsequently, followed by memory checking, correct output, and finally coverage.

Program return values

Background. When a program finishes, it returns a value indicative of success which is the argument to `exit()` (or, for IVI library users, `ivi_exit()`. This value is available in bash as `$?` and in tcsh as `$status` immediately after the program has finished. A program can attempt to return anything it chooses, so we have to rely on conventions, which are a complex mixed bag. Exit values generally get squashed into an unsigned integer so anything outside 0-255 is risky at best. Typically, a program will return 0 on success, a small positive integer for some kind of error, and 128+n for exit due to signal number 'n'. An exit value of 124 is what the program 'timeout' returns on timing out, and 255 is often what you get when you try to return something outside 0-255. By default, the test scripts will regard any return value greater than 123 as failure (but this can be circumvented). We currently define `EXIT_CANNOT_TEST` and `EXIT_BUG` to be 198 and 199 respectively, but it is possible that we need to change this in the future (hence use the symbol not the number in code). If something goes wrong, your program should return `EXIT_FAILURE`, which is defined as 1, to report a user or system problem (e.g., a needed file does not exist, a command entered by the user is invalid, out of memory, etc.,), but `EXIT_BUG` if a bug is detected (or call `ivi_abort()` which will lead to a return of XXX). Examples include a function that is documented to return positive numbers but gave you a negative one. Make sure you understand the difference between the two kinds of failures.

Return code handling in test scripts. Generally, a successful run will return 0, but we allow for testing that a program returns an error status when appropriate. For example, you might want to test the your program's error handling when a file does not exist. So, we need to specify the expected return value of a program as part of a test case. Usually a return code of 124 or higher is a failure due to a bug that cannot necessarily be predicted, so the test scripts assume failure if that occurs. However, it could be the case that the purpose of program is to crash, or you might want to test bug handling, or you might want to test the test scripts. Hence the test scripts provide a mechanism to declare any return value as success or failure (see below).

Return codes for test programs. If the above is confusing, here is the **executive summary**. Usually your test program should call `ivi_exit` with `EXIT_SUCCESS` on success, `EXIT_FAILURE` if you are reporting a user or system error, and `EXIT_BUG` if what is being tested fails. In addition, if you use some of the robust macros described above, then failure might come in the form

of a call to `ivi_abort()`.

Crashing

If a test run results in a segmentation fault or a bus error, then the return code will be XXX and XXX respectively, and the test scripts will generally assume that it is a failed test. This can be circumvented, as, believe it or not, there are reasons to write code whose job it is to crash. For example, how does one test that the test scripts do the right thing when a program crashes?

Internal checks

Robust code will often have a number of internal checks which are often in the form of asserts (convenient macros wrapping asserts are defined in XXX and XXX). Typically these will lead to an abort, which implies a return code that is typically treated as a bug.

In addition, a very important way that we test many functions is to compute a result on (often) randomly generated input, and check that it is what we expect. If it is not, then the test program should return `EXIT_BUG`. This need not be immediate, as we might want to check many cases and report how many failed.

Methods to check that the output is what we expect that you will see in the test programs include:

1. Checking that two or more different methods to compute the result give the same answer. For example we might have an old, slow, but reliable way to compute a result, and we want to check that a new, fast, perhaps multi-threaded way gets the same answer within tolerance.

2. Many functions are naturally checkable. A list returned by a sort function can be checked for being in order. Matrix inversion can be checked by matrix multiplication. An equation solver for $f(x) = 0$ can be checked by evaluating that the returned value in fact evaluates to zero (within tolerance).

3. We can use a different program such as MATLAB to provide answers to file inputs.

For all these cases, the difficult part is often deciding what a good tolerance is. This might require understanding something both about the algorithm and how floating point arithmetic works in computers. However, this is something worth knowing about.

Memory checking

In development mode, the IVI library automatically does some memory checking, so during program development you will already have been warned about some basic memory issues. In addition, on linux systems where valgrind is installed, the test scripts run programs using it for additional checking. A successful run of a test run requires that valgrind does not detect any errors. However, since valgrind can be slow, its use can be disabled as appropriate while developing test cases. The test script reports two kinds of valgrind issues: memory leaks and memory errors. In either case, you will be pointed to the valgrind log file for further information. The output of these files can be a bit daunting if you are not used to them. Usually you use them to find out where the error is, and then check the code, rather than decipher the messages, although they can be useful. You should be aware that valgrind can locate your issue better in development mode (because we compile with the '-g' flag), so you should not delve into valgrind errors for production mode unless they do not occur during development mode.

Correct output

For our current purposes we exclude fundamentally interactive programs (e.g, video games).

A program is defined by the relationship of input to output. Input includes relevant environment variables, program arguments, standard input, and any named files. Output includes the return value (already discussed), standard output, standard error, and any named files. Our testing scripts allow for the possibility that a named input file is also a named output file (i.e., the point of the program is to update a file), but this has not yet been tested (TODO). Our testing protocols thus require a human in-the-loop to verify that this relationship is in fact correct, which is possibly the main weakness in any testing system. However, many tests do not have any output other than the return value which is easily checked, and you will be warned if it is not 0 (i.e., if you are establishing a test program that is meant to return

non-zero, you will be reminded that this is unusual).

Code coverage

UNDER CONSTRUCTION

6.2.2 Testing scripts

Automated testing is done using the build system with targets 'test' or 'regress'. The build system achieves this using the script `test_program` (which lives in `src/Make/scripts`), which uses `test_program_2` to do the heavy lifting. The script test_program accepts the case '-i' to interactively create test cases (see below). The other case options are used by the build system for automated testing.

Directory structure

Library function test code is generally found in a 'test' sub-directory of the library component directory (e.g, `src/lib/l/test`), but testing can be done in any directory. The distinguishing feature of a directory used for testing is the existence of one or more programs (i.e., code file with main()) and a sub-directory 'test_input' with the following structure:

```
test_input/[program]/[test_case]
```

The program is the test program name, and `test_case` is any convenient label, with the following advisories. The label NULL is typically a system created test case with an empty directory, and labels of the form DD (where D is a digit) are often system created ones from running `./test_program -i`. Also, test cases with suffixes '.slow', '.broken', and '.fails' are omitted when associated environment variables are set. However, there are no restrictions on tests case labels, and the DD form is a fine default.

The test_case directory holds any needed input files, and any of the following special files (none need exist, and only the first two are commonly used):

```
ARGS STDIN TIMEOUT SKIP_TEST ACCEPT_RC REJECT_RC TEST_FAILURE_TESTING
```

`ARGS` is a single line containing input arguments. If it does not exist, then no program arguments are used. `STDIN` is a file used as standard input. If it

does not exist, then `/dev/null` is used. If `TIMEOUT` exists, it is a single positive integer limited the testing time in seconds (NOT TESTED!). If `SKIP_TEST` exists, the test is skipped. If `ACCEPT_RC` exists, it is a single line with one or more integers defining return codes that we will assume are successful runs. If `REJECT_RC` exists, it is a single line with one or more integers defining return codes that we will assume are failure runs, Finally, if `TEST_FAILURE_TESTING` exists (content is irrelevant), then the sense of the test is reversed (success means failure).

The result of running tests is stored in a directory 'test_runs' with the following structure.

```
test_runs/[program]/[test_case]/"test"|"regress"/[code_type]/
```

These directories are maintained by the test script. However, you will often look at them as they store the results of the test including output files, return codes (in the file RC), testing logs, valgrind logs (when applicable), and the file `FAILED` if the last test failed. They will also contain a copy of relevant input files, both for convenience, and to support programs whose purpose is to modify an input file (NOT TESTED!).

Testing protocol

The basic cycle for testing is

1. Write a test program.
2. Create test cases and check that the program is doing the right thing.
3. One things seem to work, put the files where they need to be (if they are not already there) and then run tests (i.e., `make test`)
4. CHECK the results and touching the file 'output checked' in the run directory. If you have done the above, the results should be OK. However, this step is consistent with retrofitting existing tests.
5. Check that regression testing works with `make regress`.
6. Switch to production mode (assuming that you were using development mode so far) and do the regression test again.

7. Put the test into the SVN repository (git support to come soon). For SVN, you can use `make test_svn` to automatically add files that have not been added yet but are needed for testing. Then do `svn commit`.

8. If you have developed this on a mac, then you need to log onto one of linux servers (e.g., laplace), update, and then do a `make regress`, which likely will do a (`make test`) first. This step checks that the code works on linux and that it passes valgrind's tests. (Valgrind is not ready for prime time on macs, and may never be).

9. Re-run regression tests during further development, and also automatically do so using nightly runs and/or as the result of pull requests. Not that when you do regression tests on programs, all regression tests on libraries that it depends on will run (SOON!).

The build system will build code it needs to run tests. But it might be helpful to remember that 'test' does not depend on the code (so it will not rebuild the code if the executable exists, and won't rebuild just because the source code has changed), whereas 'regress' does rebuild the executable if it is out of date with respect to the source code. This behaviour is intentional but subtle. If test were to depend on the code, then the test would be out of date when the source changes, but 'test' merely establishes a relationship between input and output. On the other hand, 'regress' checks that the relationship persists as the code base evolves.

Summary of make targets related to testing

The target 'test' builds the test and even if things are working, the build will fail until you touch 'output_checked' for the test. The script will provide the command for doing so but be sure that the results is OK. Nearly everything can be automated, but **not this step!** Once this is done, the test will not run again on that until forced. The target 'test_svn' will add files to SVN as needed. (Support for git will come soon). The target 'regress' runs the tests provided that the code has changed, and is a function of compile settings and the machine you are running on — if the code is on the IVILAB shared file system, then the test will run separately on the CPU servers that are sufficiently different

Under normal circumstances, you will only need the above targets.

KOBUS SAYS: The following should be rewritten after we have more use cases about developing tests. Most of our experience with this system is retrofitting existing tests, not developing new one. ←

When tests fail, you generally will change the code to make things right. However, to help with script debugging, or setting up tests, the following targets might be useful on occasion. Since once 'output_checked' exists, the test will locked in and not necessarily run again, to rerun it (perhaps to update output files after debugging) you might need to force rerunning with the target 'test_clean'. If major work is needed, you might want to invalidate the test while debugging with the target 'test_invalid', which removes 'output_checked' files, which will force rerunning until 'output_checked' is reinstated. The target 'regress_clean' can force rerunning regression tests, but it is not clear how useful this is other than to debug scripts. The target 'test_very_clean' completely kills the test by removing the appropriate directories in 'test_runs'. It is recommended that you avoid doing so if possible, due to SVN tracking. If you do need to use this, or any other way to kill a test, then you should first be up to date with respect to the SVN repo, and immediately follow the removal with a commit.

All the just mentioned targets can be followed by a period, and then the name of a program, to run tests on only that programs (e.g., `make regress.2D_arr`). The order of the fields is generally exchangeable (e.g., you can also use `make 2D_arr.regress`).

Manipulating the testing behavior

In addition to the special input files and test case suffixes mentioned above, testing can be manipulated with the following environment variables.

```
CREATE_TEST_DIRS ASSUME_OUTPUT_IS_OK ADD_TESTS_TO_SVN SKIP_VALGRIND
SKIP_SLOW_TESTS SKIP_BROKEN_TESTS SKIP_FAILING_TESTS
```

The first one requires a `make depend_clean` before the `make test`[2], and tells `make depend` to create a NULL test case for programs that don't have any test cases. `ASSUME_OUTPUT_IS_OK` is for retrofitting existing tests, and declares that the output need not be checked. The third one asks the script to execute

[2] `make test` will check that `depend` is up to date, but we want to force a rebuild of `depend` with the condition that this environment variable is set — this is a bit of a hack awaiting use cases to see if a more principled solution is justified.

`svn add` instead of providing suggested svn command lines to be manually entered[3]. If `SKIP_VALGRIND` is set, then valgrind memory checking is omitted. The last three environment variables provide for skipping tests based on case name suffixes, and are meant for developing tests, and documenting which ones are broken or fail. (Broken means the test program needs work, fail means the function being tested needs work). The broken and failing cases might also have the file `SKIP_TEST` in the input directory so that automated testing does not continually report known issued.

Support for creating tests

The few that have used the testing scripts before might notice that this is a new feature as of summer 2017. Consequently, this feature has not been tested much. Please help work out the bugs!

The test script can help construct test cases for programs which read from stdin, but are typically tested by typing input instead of providing a file of input to feed in via stdin. There are many such programs scattered in the library test dirs in sub-directories named 'interactive'. To create a test for such a program, go to the program directory and enter:

```
test_program -i [program] {args}
```

(This assumes that `$SRC_DIR/Make/scripts` is in your path. Otherwise you need to use `$SRC_DIR/Make/scripts/test_program`.) This will make a test case for you with case_label of the next available DD, where D is a digit. The test script will capture input from the terminal, and output to stdout and stderr, as well as the return code, and put it into the corresponding directory in test_runs. If you like your test, add an `output_checked` file, and do a `make test_svn`. If you don't, delete the test case directory.

The test script is not applicable to all programs. It assumes that the program can get all of its input from program arguments, named files, and stdin. In particular, directly reading from the terminal will not work. Hence `TERM_BUFF_GET_LINE()` (macro for `term_get_line()`) cannot be used. For this reason, we have added `STDIN_BUFF_GET_LINE()` (macro for `stdin_get_line()`) which switches from terminal read to non-terminal read depending on

[3]The interaction of svn with the test scripts is a new feature. I expect that `make test_svn` will become the preferred way to add tests to svn, in which case we might want to reduce the flurry of output

whether stdin is a terminal[4].

Finally, the script assumes that output to stdout is flushed after every line when the program is writing to a pipe. Since this behavior is the default for our library routines such as `pso()`, `ivi_frwite()`, and `put_line()`, and you would typically be using the test script with the library, generally there is nothing to do to satisfy this condition. But, if you are using certain other routines, such as raw `printf()`, then you would need to use a `fflush()`.

6.2.3 Documenting testing

For many years the documentation builder has simply inserted disclaimers saying that routines are not adequately tested. This is generally true, but not very informative because some routines are tested better than others. While we will continue inserting some kind of disclaimer, it is also worth tracking how well routines are tested, if only to hunt downs ones that are not well tested. To do so, we will add the following field to the routine header comment blocks that store the documentation for C routines:

```
Testing level: N
```

where N is a digit with the following meaning:

- 0 — It is possible that the code has never been executed. Such code should begin with the macro UNTESTED_CODE().
- 1 — The code compiles without warnings and has been executed enough, either in a real program, or in a toy test (e.g., like many programs in test/interactive), to merit removal of UNTESTED_CODE().
- 2 — At least some automated testing of the code is in place, valgrind checks pass, and there are asserts where sensible that all pass when testing.
- 3 — The automated testing has complete code coverage, and is considered thorough with respect to corner cases by one programmer. The tests pass on multiple machines.

[4]The routine `term_get_line` provides extra functionality such as input line editing and is nice to have when using a program interactively

- 4 — The code has been vetted by more than one programmer both by reading the code, and writing additional test cases as needed. We have tiny bits of code that come close, but nothing interesting that really qualifies for a 4.
- 5 — The code cannot fail. Ever. Nirvana. We have no such code.

To state the obvious, reaching level N implies reaching all levels less than N.

We will use a similar mechanism to add the testing level into the documentation for C++ routines, which are extracted from the code using `doxygen`. (UNDER CONSTRUCTION).

6.2.4 Profiling

UNDER CONSTRUCTION

6.3 GPU

6.3.1 Using the GPUs on Laplace

In this section we talk about the GPUs on laplace and the instructions to efficiently and safely use them. There are two Geforce RTX 2080 GPUs on laplace each of which has 10989 MB of memory. Cuda, CuDNN, Tensorflow 1.13.1 and Keras 2.2.4 are installed and tested on laplace in order to use the GPUs appropriately.

One big concern for us is to be able to run two different tasks in parallel at the same time on the two GPUs. Based on some initial experiments, once we run a Keras task on the laplace using GPUs, Cuda refuses to start another task even if one of the GPUs is idle because it takes all available GPUs. To overcome this issue, we found an environment variable that Cuda uses to find the visible GPUs and use them. Since we have two GPUs available on laplace, at the beginning of our Keras scripts, we need to specify the one GPU to run the task on, using the following command:

```
import os
os.environ["CUDA_VISIBLE_DEVICES"]="0"
```

The number zero should be replaced with one if the second GPU is free.

Chapter 7

Writing and Presenting

Writing and presenting are key skills that we all need to continuously work on. The job is never done. A key component is to get critical feedback and to learn from it. For example, in the IVILAB we take preparing for talks very seriously, and the feedback given is useful both to the presenter and the audience. While the experienced presenters will grab priority, it is also important to have as many points of view as possible about what works and what does not.

In spring 2018 we introduced our own acronym about the main issues in writing and presenting: PASS. PASS stands for purpose, audience, story, and structure. While "story" and "structure" have some overlap, redundancy here is OK, and PASS is a better acronym than PAS.

The rest of this chapter is under construction. In the mean time, refer to these two aging documents for tips on writing and presenting respectively:

http://kobus.ca/teaching/professional/writing_tips.pdf
http://kobus.ca/teaching/professional/preparing_talks.pdf

Appendix A

Boot Camps

A.1 Summer 2018 — a first draft of a group study version of boot camp

A.1.1 Introduction

A number of times in the past decade the IVILAB has run *boot camps* on various getting started issues, followed by IVILAB style C/C++. These have been useful, even successful, but success comes at a steep price of instructor time (typically IVILAB grad students and PostDocs). In addition, we have assumed that everyone should go through all of the boot camp material at the same fast pace (hence the name boot camp). This one size fits all mentality has led to insufficient overall commitment to the process — basically homework was not given enough attention. Hence, starting 2018, we will test out developing a set of modules, based on previous boot camps, that can be studied by interested groups or individuals at their own pace, and need not be limited to summers. Instructors will not arrange sessions or set schedules, but will be happy to answer questions over email, and meet on occasion when advantageous.

A.1.2 Who Should Undertake this Study

This activity is strongly advised if you will be working on the IVILAB C or C++ code base for your research. If your project is restricted to C, then completing the first half might suffice. This activity might also make sense for

those interested in learning or advancing C/C++ skills, or even programming in general. However, the plan of study is geared towards using and improving the IVILAB C and C++ code.

A.1.3 Time Commitment

While designed as self-paced, one should acknowledge one's goals and time availability to learn the material, which requires a lot of time for "hands on" activities such as programming. Depending on your background and focus, most modules will take at least 10 hours. Doing a module per week reflects the original boot camp intention, but it may be better to do a good job with each module, rather than keep up an artificial pace.

A.1.4 Prerequisites and Responsibilities

Ideally, you have studied at least one programming language before this one. Previous experience with C is not needed, but those that are new to C will need to budget extra time for the module that goes over basic C. Other than knowing at least one programming language, the main prerequisites are the same as those for updating notebooks, i.e., basic command line usage, text editing, SSH, SSH keys, and SVN (see §2). You should update your mentors on your progress through your notebook. You should complete preceding modules before undertaking a new one, except in consultation with your mentors. Code written for this study (except when you are contributing to the IVILAB code base) should be in SVN under src/-bootcamp/YOUR_NAME/module_NN. Finally, to respect the email boxes of those that are not currently going through the material, we will deal with boot camp issues using a specific mailing list (for now, we will reuse the existing one, ivilab-code).

A.1.5 UNDER CONSTRUCTION

There will be many errors and deficiencies. Please help improve the modules. Report any issues as soon as possible and/or fix this document (manifesto file boot.tex).

A.1.6 Philosophical Foundations

An important philosophy underlying IVILAB coding is that we build the code base together. This means that you should use what is in place, or fix it, if it is broken or out of date. While no code base is perfect, it gets better if it is used, which exposes issues, and improved in preference for simply trying to get the immediate job done, which is how most programmers operate most of the time. If we should be doing something differently, we prefer to know about it, and have a discussion about it, rather than be ignored.

In addition to the above, the IVILAB coding conventions are particular about error messages (they should be good), memory management (minimize calls to the storage allocator), debug code versus production code, efficiency where it might matter, and unit testing.

A.1.7 Texts and Other Resources

LaTex

If you are unfamiliar with LaTex, you will find many good online sources for learning, with this beginner-friendly tutorial being one of the best for explanation and example.

For macos, it is suggested that you install MacTeX, which is bundled with TeXShop. However, you will be doing most of your IVILAB work in VIM.

You can open and edit your files using the following command in your terminal: `vim filename.tex`. After you edit your text you will need to build your pdf.

Though you can use any pdf viewer, such as SKIM, using the command `latexmk -pdf filename.tex` to build the pdf. Another way is to simply use your standard pdf viewer on your laptop that comes with the Catalina OS. After editing with vim, use these two commands to build your pdf:

```
pdflatex filename.tex
open filename.pdf
```

For the Ubuntu, type this command to install:

`sudo apt install texlive-latex.extra`. You may or may not need to install an extra font package or two. When you build your pdf, you'll find out soon enough. To build, you can install your favorite pdf viewer or use the standard evince that comes with the ubuntu:

```
pdflatex filename.tex
evince filename.pdf
```

C++

We will structure our study around Bruce Eckel's book, "Thinking in C++", which is available on-line free of charge. Other resources include: XXX.

A.1.8 Module one

1. Read chapter one of Eckel's book which should prompt you to start thinking about the craft of programming and object oriented design. While we need not necessarily subscribe to everything he says, you should be thinking about these things. Also, while C does not explicitly support object oriented programming (which motivates C++), the IVILAB (KJB) C libraries and processes embody much of the same philosophy. Hence, chapter one of Eckel's book is a good place to start.

2. If you have not already done so, set up your summer notebook (see §2), which entails learning the basics of SVN.

3. You will either be working on your own laptop running linux or Mac OS, or the IVILAB computer v11. In either case, make a directory "src" wherever you like (your home directory is fine).

4. Check out the key IVILAB directories into your "src" directory with:

   ```
   svn co svn+ssh://vision.cs.arizona.edu/home/svn/src/Make/trunk Make
   svn co svn+ssh://vision.cs.arizona.edu/home/svn/src/lib/trunk lib
   svn co svn+ssh://vision.cs.arizona.edu/home/svn/src/include_after/trunk include_after
   ```

5. Add the directory src/Make/scripts to your path. You can check that it is working properly with `which kjb_add_makefiles`.

6. Make a directory in SVN to hold your bootcamp exercises (except ones that contribute to the code base) by:

   ```
   svn mkdir svn+ssh://vision.cs.arizona.edu/home/svn/src/bootcamp/YOUR_NAME
   ```

7. Subscribe to the directory just created. You can map it into any directory name you like. The example below calls it "bootcamp"

```
svn co svn+ssh://vision.cs.arizona.edu/home/svn/src/bootcamp/YOUR_NAME bootcamp
```

8. Make a directory for this module, which will provide a start of the organization for the rest of the study. In your bootcamp directory (if you called it that), create an SVN directory for this module called "module_01" by:

```
svn mkdir module_01
svn commit
```

9. In your module_01 directory, create (or copy) a "hello world" program in either C or C++. If you need inspiration, you could peek into the first little bit of Eckel chapter two.

10. Compile your "hello world" program (perhaps using g++) and check that it works.

11. KOBUS SAYS: Need to check the minimal instal set. ←

    For the next step, you should be able to run the following programs: tcsh, sed, grep, vim, make, makedepend, gcc, g++, XXX. If you are not using an IVILAB computer, you may need to install some software. See §5.1.2 for a list of packages to install and how to get them. For this module, you can skip the entries `libgsl` and below.

12. KOBUS SAYS: Add a pointer to the build system documentation once it is more interesting. ←

    Create a build system for your code in your Module one directory using the script `kjb_add_makefiles`, and verify that you can build your hello world program by typing `make`.

A.1.9 Module two

1. Read chapter two of Eckel's book which is about getting started with C++ without worrying about the details. One should be able to do the programming assignments largely by following the examples in the chapter.

2. Add a directory "module_02" in your boot camp location, and be sure to add it to SVN.

3. Begin to get familiar with the IVILAB code formatting guidelines (see §6.1.3) in specific, and §6.1.2 through the beginning of §6.1.7, in general) so that you can do the next activity in approximately the IVILAB coding style.

4. Do about half (for or five) of the exercises at the end of chapter two. The solution to each one should be an appropriately name file in module_02 that should also be added to svn. Each time you add any code, you should be able to build / rebuild by simply typing "make". In particular, notice that the build system rebuilds makefiles when you add (and delete) code files. This is a feature of the IVILAB build system. Without it you would have to know when to manually rebuild makefiles, or even worse, manually edit makefiles. If this automatic rebuild does not work for you, then: 1) make sure that you commit your files; and 2) let Kobus know.

A.1.10 Module three — crash course in C

1.

A.2 Summer 2017

A.2.1 Week 1

Introduction to Bootcamp, mailing lists, SSH

A.2.2 Week 2

More SSH - getting everyone set up

A.2.3 Week 3

- Introduction to SVN - checkout, add, update, merge.
- Checkout main support libraries, documentation, and 2017 bootcamp folder.

A.2.4 Week 4

- Mon: Your first KJB program (read and write vectors from file)

```
#include "m/m_incl.h"

int main(int argc, char** argv) {
  kjb_init();
  Vector* vec = NULL;
  read_vector(&vec, "test_vector.txt");
  write_row_vector(vec, NULL);
  free_vector(vec);
  return EXIT_SUCCESS;
}
```

- Thu: Introduction to pointers, more KJB programs - Matrix operations,
  - Update the SVN bootcamp repo (`svn up`)
  - Exercise 1: Copy the file `bootcamp/test_matrix.txt` to your directory. Write a C program called `Ex4_1_C.c` that adds the rows of the matrix in `test_matrix.txt`. The output should be the vector `12 15 18`. Use the kjb C library like we did for the example last week.
    Solution:

```
#include "m/m_incl.h"
#include "m/m_mat_io.h"
#include "m/m_mat_arith.h"
#include "m/m_mat_stat.h"

int main(int argc, char* argv[]) {
  Matrix* mat = NULL;
```

```
    kjb_init();
    read_matrix(&mat, "test_matrix.txt");
    write_matrix(mat, NULL);
    Vector* vec = NULL;
    sum_matrix_rows(&vec, mat);
    write_row_vector(vec, NULL);
    return EXIT_SUCCESS;
}
```

A.2.5 Week 5

- Mon: C++ programs, image manipulation.
  - Exercise 2: Make a copy of `Ex_4_1_C.c` called `Ex_4_1_CPP.cpp` and make it run. Note: When you add a new program that has a `main()` function in it, you need to run the `kjb_add_makefiles` script again. You can call kjb C library functions from C++ functions. What extra steps do you need to make the program compile as a C++ program? (Hint: namespaces - look at the Doxygen documentation for the C++ libraries).
  - Exercise 3: Copy the file `bootcamp/mona-lisa.jpg` to your directory. Now write a C++ program to draw a red circle around Mona Lisa's face. Solution:

```
#include <i/i_float_io.h>
#include <i/i_draw.h>
#include <m/m_incl.h>
#include <iostream>

int main(int argc, char* argv[]) {
    kjb_c::kjb_init();
    kjb_c::KJB_image* img = NULL;
    kjb_c::kjb_read_image_2(&img,"mona-lisa.jpg");
    kjb_c::image_draw_circle(img,150,250,100,1,255,0,0);
    kjb_c::kjb_write_image(img,"mona-lisa-modified.jpg");
    return EXIT_SUCCESS;
}
```

- Thursday: Structs and classes

  - Copy the files from `in_class_examples` to your bootcamp directory.
  - Build the two executables `test_Vector3D` and `test_Animals` using the KJB build system, and look over the source code.
  - Create a class for your favorite animal (that inherits from the Animal class), and implement a constructor for it that sets its speed. Compare how far your favorite animal can go in 5 seconds (vs a Cheetah)

A.2.6 Week 7

- Mon: C macros, intro to KJB error handling macros
- Thu: KJB error handling explained in more detail, how a stack of exceptions is created, KJB lib/ naming conventions and structure.

A.2.7 Week 8

- Mon: Virtual functions, templates
  - Exercise 8.1:
    * Go to the directory where you have the `Animals.h, Animals.cpp` and `test_animals.cpp` files.
    * Declare the function `char* animal_type()` for both the Animal and Cheetah classes in `Animals.h`. This function should return `"Animal"` for the Animal class and `"Cheetah"` for the Cheetah class. Implement this functionality in the `"Animals.cpp"` file.
    * In your `test_animals.cpp` file, write a function that takes an Animal as an argument (passed by reference, not by value), and prints out the type of the animal (as obtained from the `animal_type()` function.)
    * In the `main()` function, declare and initalize an Animal named Alice and a Cheetah named Bob. Use them as arguments to the function you wrote in the previous step to print out their types. What do you expect will happen?
  - Exercise 8.2:

* Here is an example of a template function:

```
template<class T>
T add_numbers(T x, T y) {
    return x + y;
}
```

* The arguments x, y can be of type `int, float, double`, and you would get the same result. You can see some obvious utility in this - you don't have to write three separate functions such as: `add_ints, add_floats, add_doubles`. In a way, templates are similar to preprocessor macros, but more powerful.
* Now, copy the file `Vector3D.h` in your directory to a new file called `Vector3D_template.h`.
* Convert the original class (which only accepted arguments of type `double`) to a template class which can hold other kinds of numbers. To do this, add the line `template<class T>` above the class declaration, and replace all instances of `double` with `T`. Also note that with templates, the declaration and definition of the functions cannot be in separate files, so you will need to define the functions in `Vector3D_template.h` itself. Now include `Vector3D_template.h` instead of `Vector3D.h` in `test_Vector3D.cpp`, and try running the program. To create a Vector3D of doubles called myVector, you can do something like

```
Vector3D<double> myVector(1., 1., 1.);
```

Appendix B

Documentation for build scripts

This appendix is built from the man files, which themselves are build automatically from text within the script files. Not all scripts in src/Make are documented. If you would like to document one that does not have documentation (yeah!), then use the comment block of one that is documented as a template (example_script).

We divide the scripts into two groups, and list them alphabetically within the groups. The first group are scripts that one might want to use in general. The second group are of interest mostly for working on the build system itself (it would be rare to use them for anything else.

B.1 Generally useful build scripts

B.1.1 example_script

We do not include the documentation for example_script here due to its characters for on screen underlining. Either do a "man example_script" or look at its source in the Make/scripts directory.

B.1.2 export_machine_vars

```
export_machine_vars(1)                                    export_machine_vars(1)

IDENTIFICATION
    A script to get some compile vars into bash scripts.

DESCRIPTION
    A script to get some compile vars into bash scripts. In the bash scripts one
```

```
    uses:
        eval `path_to_this_file`
    e.g.
        eval `/home/kobus/src/Make/scripts/export_machine_vars`
    This sets:
      MACHINE, UC_MACHINE, UC_OS

    Currently we only export the above variables set by init_machine for Kobus's
    courses so that students can try the ivi library without using the source.

    However, if we need things like IVI_CC and OBJ_DIR, we could extend this
    script, or write a similar script, that consults init_compile as well.

AUTHOR
    Kobus Barnard
```

B.1.3 fix_mac_dylib_location_ids

```
fix_mac_dylib_location_ids(1)                    fix_mac_dylib_location_ids(1)

IDENTIFICATION
    A script to reset the location ids of mac dylibs and to patch lib refs

DESCRIPTION
    On mac OS, dylibs have strings that declare where they should be (e.g., the
    path where the linker should look for them). If you move them, then you have
    an inconsistency that can cause you grief, as the library might not be found
    at run time (even though you tell the linker where it is), or the wrong
    version will be loaded at run time. This script fixes this, if it is
    fixable, which is why we report a before and after.

    This script needs to be executed in the directory where the libraries are,
    even if they are links.

    In June 2019, I added code that also changes absolute reference to any dylib
    named in this directory, to be the one in this directory. This addresses the
    problem that you can move lib files (e.g., lib/libboost*) to somewhere else
    (e.g., lib/boost_1_54), but its components (e.g.,
    libboost_filesystem-mt.dylib) can have pointers to other components (e.g.,
    libboost_system-mt.dylib). Since these are encoded within absolute paths, we
    need to fix the absolute paths since we know we have moved the components
    that have those stale absolute paths.
```

```
IMPORTANT
    If this script changes anything, it is quite likely you will want to do (as
    root)
        update_dyld_shared_cache
    followed by a hard reboot (shutdown, not just restart).

    You can use 'otool -L [program]' to see if your program agrees where to find
    the libraries. Another trick is to at the flag "-t" to HACK_AFTER_LOAD_FLAGS
    in BUILD_ENV.${USER} which will tell the linker to tell you where it found
    stuff. This flag causes no harm, it just creates a lot of output, so you
    will want to remove it from BUILD_ENV.${USER} when you are done debugging.
```

B.1.4 init_compile.sh

```
init_compile.sh(1)                                          init_compile.sh(1)

IDENTIFICATION
   Provides the IVI build environment variables to bash users

DESCRIPTION
   This script provides the IVI build environment variables to bash users. It is
   used by the bash versions of the build aliases (e.g., set_prod). It also
   adjust your PATH variable as the build changes. Specifically, it also
   pre-pends the LD_OBJ_DIR to PATH. In this way, it is like the tcsh script
   init_compile_full.

   This script must be sourced in bash (or its contents copied for use by bash).

   This script assumes that IVI_SRC_PATH points to a directory which has Make as
   a subirectory that has the init_compile script in it, and also scripts as a
   sub-sub directory. (The standard layout in SVN of src/Make). Since the
   default location of this script is ${IVI_SRC_PATH}/Make/scripts, it seems
   safe to assume that IVI_SRC_PATH can be set as it is implicit in calling this
   script anyway.
```

B.1.5 get_readable_mod_time

```
get_readable_mod_time(1)                              get_readable_mod_time(1)

IDENTIFICATION
    Prints path modification time in human readable format
```

```
DESCRIPTION
    This script prints the modification time of its argument in human readable
    format in one second resolution. Its main use is for debugging. Kobus also
    has a script "get_mod_time" that returns higher resolution modification
    times as integers that are suitable for comparisons. If needed, that script
    can become part of the Make/script scripts.

    We implement this versiion using "/bin/date -r" when avalable, but provide
    sensible error messages if the arugment is not an existing path. The program
    "date" is seriously broken as the flag "-r" has two meanings, and if the
    argument does not exist, it will try to convert it to an integral value for
    seconds, and when that inevitably fails, it will give you an obtuse error
    message.

AUTHOR
    Kobus Barnard
```

B.1.6 ivi_add_makefiles

```
ivi_add_makefiles(1)                                    ivi_add_makefiles(1)

IDENTIFICATION
    A script to add ivi style Makefiles the current directory

ARGUMENTS
     No arguments are required. The argument '-p' forces adding program
     makefiles. The argument '-l' forces adding library makefiles.

DESCRIPTION
    This script adds ivi library style Makefiles to the current directory. In
    the absence of a "-p" or "-l" flag, if there are no code files at all, or if
    a main routine can be found, then it is assumed that program oriented
    makefiles are wanted; otherwise, it is assumed that library oriented
    makefiles are wanted.

    If program oriented makefiles are wanted, and there exists a sub-directory
    "lib", then this script will build makefiles in any sub-directory of "lib"
    that has code.

BUGS
    As we climb up the directory tree from the current directory we are looking
    for an ancestor that has the sub-directory "Make". If we happen to back up
```

```
    through a symbolic link, then this script will probably fail.
```

B.1.7 ivi_cat_after

```
ivi_cat_after(1)                                        ivi_cat_after(1)

IDENTIFICATION
    Echo lines once a match to the argument is found

DESCRIPTION
    This script reads from stdin and does nothing until the pattern specified in
    the first argument is found, which is echoed unless the first argument is
    argument is '-a' (really "after"). The last argument is the pattern.

    This script requires assumes that "init_scripts" has been sourced (or
    P_STDERR, IVI_TAIL, and IVI_AWK have been set by some other means).

    We implement this functionality with "awk", but wrapping this capability in
    a script means we can do something else if we run across a system where this
    does not work.

NOTES
    We could implement the ivi_cat_before/ivi_cat_after functionality with
    "split", but that program does something different, and is not commonly
    used, so it may not be available.

AUTHOR
    Kobus Barnard
```

B.1.8 ivi_cat_before

```
ivi_cat_before(1)                                      ivi_cat_before(1)

IDENTIFICATION
    Echo lines up and including a match to the argument

DESCRIPTION
    This script reads from stdin and echos output until the pattern specified as
    the last argument is found, after which there is no more output. The first
    line matching the patter is is also echoed as the last line unless the first
    argument is "-b" (really before).
```

```
    This script assumes that "init_scripts" has been sourced (or P_STDERR and
    IVI_SED has been set by some other means).

    We implement this functionality with "sed" line, but wrapping this in a
    script means we can do something else if we run across a system where this
    does not work.

NOTES
    We could implement the ivi_cat_before/ivi_cat_after functionality with
    "split", but that program does something different, and is not commonly
    used, so it may not be available.

AUTHOR
    Kobus Barnard
```

B.1.9 ivi_lock

```
ivi_lock(1)                                                    ivi_lock(1)

IDENTIFICATION
    A script to execute commands so that only one instance is run at a time.

SYNOPSIS
    ivi_lock {max_time} {cmd} [args]

DESCRIPTION
    This is a simple script for locking. It trades limited capability for
    simplicity (see deficiencies below). The main justification for this script
    is that locking is not universally available. Most notably good locking
    facilities are not available in the mac OS. To get portability, we implement
    a lock_create and a lock_remove, using locking facilities that are available
    depending on environment variables. This script wraps the command to be run
    inside the create remove pair.

    The first argument to ivi_lock is the max number of seconds to wait. This
    has to be first because the command can have an arbitrary number of
    arguments.

    The second argument to ivi_lock is the command to be executed including its
    arguments. This script supports simple locking through a lockfile whose name
    is derived in one of two ways. One is based on the command and last element
    of the current path, and the other is based on a resource labeled by the
```

```
    value of the environment variable IVI_LOCKFILE_NAME.

    For command based locking (simplest), this script constructs a lock name
    based on the command and a piece of the current directory. This makes sense
    if the command is the only way that particular files get modified and/or
    might do a sequence of steps that all need to be protected, and/or actions
    needing protection are not described as modifying an obvious resource (e.g.,
    updating an SVN repo).

    However, command based locking does not support multiple copies of the same
    script running in the same directory that share a resource that needs to be
    protected. For this scenario, the set of scripts can set a specific named
    lock via the environment variable IVI_LOCKFILE_NAME. Usually this is an
    informative string (no slashes are allowed). However, if IVI_LOCKFILE_NAME
    holds a path, then that path is taken as the precise lockfile, and the
    behaviour is similar to "flock", especially if flock is the underlying
    implementation.

    If the environment variable IVI_LOCKFILE_PATH is set, this script will use
    its value as a directory to create lockfiles in. Otherwise, this script will
    use ${TMPDIR}/${USER}/lockfiles/, creating it if needed.

    If the environment variable IVI_DONT_LOCK is set, then this script runs the
    argument without any locking. The main purpose of this is to avoid deadlock
    when we effectively call scripts recursively. Normally there is a better way
    to accomplish what you want to do, but one instance where such a way is yet
    to be discovered is in repository synchronization where we want 'to' and
    'from' processes to share locks, but also call each other.

    This script checks for possible deadlock and does not lock if it thinks this
    could happen. Further if the environment variable EXIT_ON_DEADLOCK is set,
    then we exit with error status in this situation (good for debugging).

    Finally, this script relies on ivi_lock_create for handling timing out. If
    the timeout is exceeded, then we ignore the lock. However, if the
    environment variable EXIT_ON_TIMEOUT is set, we will exit with error status.

See also
     The script ivi_lock_for_make combines locking with not executing the
     command if the target was build by the process we were waiting on, or if it
     is newer than a file touched in build-2. Scripts ivi_lock_create and
     ivi_lock_remove can be used in other ways, and their documentation provides
     additional information about the locking system, including environment
     variables that manipulate it.
```

```
Deficiencies
    The current implementation locks all runs of a command operating in a given
    sub-directory, irrespective of arguments or parent directory, which is is
    not a big issue as it is simply overly conservative. If this becomes an
    issue, we can put more path components in the lockfile name.

    If IVI_LOCKFILE_NAME looks like a path, we do not add ".lock" to the name,
    but if it is a simple string without '/', then we do add '.lock'. This
    behavior might confuse some (and is easy to change), but the reasoning is
    that one should be able to have full control if needed, by specifying the
    full path, and we arbitrarily keep that same behavior for relative paths, as
    the path is now relative to where you are running things, not the
    IVI_LOCKFILE_PATH (which might also confuse). This scenario seems unlikely.

    See ivi_lock_create for more deficiencies related to reliable locking.

AUTHOR
    Kobus Barnard
```

B.1.10 ivi_lock_for_make

```
ivi_lock_for_make(1)                                    ivi_lock_for_make(1)

IDENTIFICATION
    A script to lock and build a target only once

USAGE
    ivi_lock_for_make {max_time} {target} {command} [command arguments]

DESCRIPTION
    This script executes a command under a lock, but also prevents a second
    process from doing the same thing once it gets the lock. To explain further,
    basic locking serializes execution, but once a second process, having been
    assigned the same task as a first, gets the lock, it will redo the build a
    second time. This is wasteful and confusing.

    If there is only one instant of make running, and if all dependencies are
    exposed to make, and the makefiles are correct, then neither locking, nor
    ensuring only single execution of the task should be needed. However, the
    ivi build system has hidden targets which are built as the consequence of
    make driven commands but this is not exposed to make. We also sometimes we
    run multiple instances of make either on purpose or by accident. Especially
    for hidden targets, we want to consider using this script.
```

```
    The second argument is only used for early exit. If it will never exist,
    then it has no effect, but in this case, using ivi_lock might make more
    sense.

AUTHOR
    Kobus Barnard
```

B.1.11 ivi_lock_create

```
ivi_lock_create(1)                                          ivi_lock_create(1)

IDENTIFICATION
    Creates a semaphore that blocks cooperating processes from proceeding until
    removed (using ivi_lock_remove).

SYNOPSIS
    ivi_lock_create {max_time} {label} [pid]

DESCRIPTION
    This is a simple script for locking blocks of script code that is not
    conveniently expressed as a command (otherwise use ivi_lock). It works much
    like lockfile-create, and in fact, is might be implemented using
    lockfile-create if it is available. The main justification for this script
    is that locking is not universally available. Most notably good locking
    facilities are not available in the mac OS. Hence, this script also
    implements portable structure that also works most of the time without true
    OS support.

    The first argument to ivi_lock_create is the max number of seconds to wait.
    This is to be consistent with ivi_lock.

    The second argument to ivi_lock_create is simply a label (often just the
    name of the script), which is combined with the current directory to create
    a lockfile name. At the end of the protected code, you must call
    ivi_lock_remove with the second and third arguments.

    The third (optional argument) is the PID of the script interested in the
    lock. It defaults to the PID of the process that called it (this script's
    parent process).

    If the environment variable IVI_LOCKFILE_PATH is set, this script will use
    its value as a directory to create lockfiles in. Otherwise, this script will
```

```
    first try /scratch/${USER}/lockfiles/ unless PREFER_LOCAL_LOCK is set, where
    /scratch is nominally an NSF mounted fast temporary disk on IVILAB systems.
    If it does not exist, or PREFER_LOCAL_LOCK is set, then we use
    ${TMPDIR}/${USER}/lockfiles/. In all cases we attempt to create the
    $IVI_LOCKFILE_PATH path if it does not exist, and exit with 1 if we cannot.

    if the environment variable IVI_LOCKFILE_NAME is set, then it overrides the
    default heuristics for constructing the lock file name. If IVI_LOCKFILE_NAME
    looks like a path it is used as is. Otherwise, we append the string ".lock"
    to $IVI_LOCKFILE_NAME, and use that for a lockfile in the lockfile
    directory.

    If the timeout is exceeded, then we ignore the lock. However, if the
    environment variable EXIT_ON_TIMEOUT is set, we will exit with error status.

    If the OS program "flock" is available, then we will use it for an atomic
    write to the lockfile. This can be disabled using FORCE_NO_FLOCK. We want to
    be able to disable it, because the use of flock in this script is not well
    tested.

    If the environment variable IVI_DONT_LOCK is set, then this script runs the
    argument without any locking. The main purpose of this is to avoid deadlock
    when we effectively call scripts recursively. Normally there is a better way
    to accomplish what you want to do, but one instance where such a way is yet
    to be discovered is in repository synchronization where we want 'to' and
    'from' processes to share locks, but also call each other.


Deficiencies
    The current implementation locks all runs of a command operating in a given
    sub-directory, which is is not a big issue as it is simply overly
    conservative. If this becomes an issue, we can put more path components in
    the lockfile name.

AUTHOR
    Kobus Barnard
```

B.1.12 ivi_lock_remove

```
ivi_lock_remove(1)                                        ivi_lock_remove(1)

IDENTIFICATION
    Removes the semaphore that blocks cooperating processes from proceeding
```

```
SYNOPSIS
    ivi_lock_create {label} [pid]

DESCRIPTION
    This is the script of a pair for locking blocks of script code that is not
    conveniently expressed as a command (otherwise use ivi_lock). It works much
    as lockfile-remove, and in fact, is implemented using lockfile-remove if it
    is available. The main justification for this script is that locking is not
    universally available. Most notably good locking facilities are not
    available in the mac OS. Hence, this script also implements a fall back
    method that is less than ideal, but usually works.

    The argument to ivi_lock_remove needs to be the same as for the proceeding
    call to lockfile-create (excluding its first argument).

    I might have called this script ivi_lock_destroy, but since it is similar to
    the standard command lockfile-remove, I went with ivi_lock_remove.

AUTHOR
    Kobus Barnard
```

B.1.13 ivi_guard_create

```
ivi_guard_create(1)                                    ivi_guard_create(1)

IDENTIFICATION
    Creates a file that typically is used to block recursive reentry

USAGE
    ivi_guard_create [label]

    The label argument is typically the name of the calling script.

DESCRIPTION
    This script creates a guard file that is meant to block a recursive call of
    a script that should not do that. If the guard file exists, then this script
    exits 1.

    The reason to put this functionality into a script is to standardize the
    naming and location of such files. In particular, we prefer not to make them
    in the directory that the script is being run because this changes the time
    stamp of the directory, which, while not being a critical issue, can lead to
```

```
    confusing rebuilding. Doing so thus would be a small violation of the
    principle of least astonishment. Finally, all guard files are put in the
    same location (usually something like /tmp/$USER/guardfiles) enabling
    simpler cleanup and debugging.

    Once the code that needs to be guarded against reentry is finished, the
    guard file should be removed by calling ivi_guard_remove with exactly the
    same label argument.

    This script was derived from ivi_lock_create, hence there are the following
    fancy features that might never get used.

    If the environment variable IVI_GUARDFILE_PATH is set, this script will use
    its value as a directory to create guardfiles in. Otherwise, this script
    will use ${TMPDIR}/${USER}/guardfiles/, creating it if needed.

    if the environment variable IVI_GUARDFILE_NAME is set, then it overrides the
    default heuristics for constructing the guard file name. If
    IVI_GUARDFILE_NAME looks like a path it is used as is. Otherwise, we append
    the string ".guard" to $IVI_GUARDFILE_NAME, and use that for a guardfile in
    the guardfile directory.

RETURNS
    This script exits with status 0 on success, or 1 if the guard file already
    exists, or if there is an issue creating the guard file. Each failure case
    leads to an error message for stderr.

AUTHOR
    Kobus Barnard
```

B.1.14 ivi_guard_remove

```
ivi_guard_remove(1)                                    ivi_guard_remove(1)

IDENTIFICATION
    Removes the guard file

DESCRIPTION
    This is one of a pair of scripts for setting up a guard to block a recursive
    call of a script that should not do that. This script is for removing the
    guard file once we are beyond the possible recursion.

    The argument to ivi_guard_remove needs to be the same as for the proceeding
```

```
    call to ivi_guard_create.

    Some of the details of this script, such as its name, is patterned on
    ivi_lock_remove.

AUTHOR
    Kobus Barnard
```

B.1.15 ivi_svn_rm

```
ivi_svn_rm(1)                                                   ivi_svn_rm(1)

IDENTIFICATION
    A script to remove a path that might be under svn control

DESCRIPTION
    This script removes paths either with "rm" or "svn rm" depending on whether
    the path is under svn control. Originally we tried "svn rm --force" but this
    is not reliable and can give confusing messages. We assume that init_compile
    has been sourced, and that svn is available.

AUTHOR
    Kobus Barnard
```

B.1.16 make_link

```
make_link(1)                                                     make_link(1)

IDENTIFICATION
    A script to create symbolic links with a few extra features

DESCRIPTION
    This script creates symbolic links, but setting up the call to ${IVI_LN}
    (which is usually something like "ln -s -f"). It currently serves other
    scripts, mostly for the build system, and so we assume that init_compile has
    been sourced.

    If the second argument is a link, and it is the one we are trying to change
    it to (first argument), we do nothing. This means that even if multiple jobs
    are trying to make the link, we do it only once. If the second argument is a
    link, but different from the second argument, we replace it. If it is a
```

```
    directory, we move it to a safe place with message, and create the link in
    its place.

AUTHOR
    Kobus Barnard
```

B.1.17 make_ivi

```
make_ivi(1)                                                        make_ivi(1)

IDENTIFICATION
    A script to make, test, and export IVILAB software components,

DESCRIPTION
    This script nominally makes IVILAB code distributions, but to do that, it
    typically first proves that it can build the makefile, compile the code
    under different conditions, run regression tests, and build the
    documentation. Hence it has also grown into a convenient way to check the
    state of the code base. This script does a lot of things, and has lots of
    flexibility through a number of options. For some common cases, see EXAMPLES
    below.

    Specific to making IVILAB code distributions, there are two variants. The
    argument "archive" requests a snapshot of the code for backup or archiving.
    This is perhaps less relevant now that everything is under svn.

    The second variant "export" builds code source archives suitable for export.
    This assumes that all components of the code can be distributed without
    copyright concerns. This script helps matters by including only the library
    modules needed, and stripping out code that is protected by a number of
    #ifdef's such as TEST, HOW_IT_WAS_*, REGRESS_*, NOT_USED, OBSOLETE, and so
    on.

    Without any arguments, all library modules and programs in a hard coded list
    are compiled under several conditions. Alternatively, any arguments that are
    not otherwise recognized, but are directory names, are taken as directories
    to work on. These directories must be specified as relative to a src
    directory, specifified by IVI_SRC_PATH, or found by the script using some
    heuristica. Other options can specify that only library code is processed.
    Regardless, the first thing the script does is to output the programs it
    plans to work on.

    This script typically takes a lot of time, and creates a lot of output, so
```

```
    generally it is easiest to redirect output and run it in the background.

OPTION ARGUMENTS

    archive
        This requests that distributions being processed are put into
        ${IVI_SRC_PATH}/DIST/archive, where ${IVI_SRC_PATH} is usually
        ${HOME}/src.

    backup
        This requests that distributions being processed are put into
        ${IVI_SRC_PATH}/DIST/backup, where ${IVI_SRC_PATH} is usually
        ${HOME}/src. Since "archive" and "backup" behave the same, this would
        only make sense if the backup directory is linked to a different disk.

     export
         This request that archives suitable for code distribution be built.
         They are put into ${IVI_SRC_PATH}/DIST/export, where ${IVI_SRC_PATH} is
         usually ${HOME}/src.

     export_only
         This request that archives suitable for code distribution be built, and
         that only minimal other work is done.

     export_src_tree, skip_export_src_tree,
     export_standalone, skip_export_standalone
         Exports can take two forms, and the plan is to be able to toggle them
         independently: 1) We export the relavent portion of the source tree,
         which means you get one copy of needed libraries, and subsequent
         exports can be tarred over the previous ones, preferablly with the "-u"
         flag; 2) Each requested export leads to a standalone package with an
         svn revision number in the directory name. This is the default.

     export_lib_test_dirs
         By default we do a minimal set of files in an exported package, and
         thuse we do not include test code for libraries. This option requests
         adding themn.

      skip_svn
          By default, if we are making a standalone export distribution, then we
          will include revision numbers in the exported directory names. For
          this, we need to do both svn updates and commits so that the revision
          number is correct. (For example, the revision number of a program can
          depend on having library code commited.) This option requests
          proceeding as though svn does not exist. This option currently only
```

```
     has an effect when we are exporting standalone packages, but we may
     use svn more in the future. For example, assuming good internet, it
     might be the easiest way to collect the needed files. If you are using
     this script without internet, and do not use this option, then you
     will get errors that can be made non-fatal with the keep_going option.

skip_export_doc
    If we are making an export archive, then it won't have pre-built man
    pages along with it.

keep_going
    By default, this scripts terminates when there is an error, including
    build errors and regression testing failures. If this option is set,
    then the script will continue if possible, and exit with 1 if any
    errors were encountered along the way.

IVI
    This builds code and tags (depending on those options) in lib/IVI.
    Different than the option "lib" we are not focussed on building an
    exportable version. We do this building relatively early on, which also
    allows library Makefiles and code to be built faster due to more
    parallelism, provided that we do not clean.

IVI_CPP
    This builds code and tags (depending on those options) in lib/IVI_cpp.
    Different than the option "lib_cpp", we are not focussed on building an
    exportable version. We do this building relatively early on, which also
    allows library Makefiles and code to be built faster due to more
    parallelism, provided that we do not clean.

lib (also implied by "library")
    This builds an exportable version of the C library code, i.e.,
    libIVI.a.

lib_cpp (also implied by "library")
    This builds an exportable version of the C+ library code, i.e.,
    libIVI_cpp.a.

ctags (also "tags")
    This requests doing ctags. This is not always done by default, but
    options that build distributions (e.g., archive, export, lib) will also
    set this option.

skip_ctags (also "skip_tags")
     Forces no ctags.
```

```
doc
    This requests building documentations. Like tags, this is not always
    done by default, but options that build distributions (e.g., archive,
    export) will also set this option.

skip_doc
     Forces no documentation to be built.

regress (also, "regress-1" or "regress_1")
     Do regression testing.

 clean_regress
     Does "make regress_clean" in every test directory processed.

 skip_regress
     Force no regression testing even if it were to be done. If test
     directories are processed, then regression testing would normally
     force the makefiles to be built. This option disables that. Of course,
     makefiles that are out of date will be built regardless of this
     directive.

 skip_depend
     This disables forced makefile rebuilding

 skip_depend_lib
     This disables forced makefile building in the library directory. If
     you are most interested in testing builds at the program level, this
     can save time.

 skip_depend_lib_sub
     This disables forced makefile building in library sub-directories. If
     you are most interested in testing builds at the program level, this
     can save time.

 skip_export_depend
     This disables forced makefile building while the export version is
     constructed. This is mostly useful for debugging. Normally, one would
     want this test if the code was really being exported.

 code
    This requests building code (compiling). The default is to do so, so
    this option is not necessary, but it is available for future changes,
    and including it in calls to make_ivi in scripts can make the intention
    more clear.
```

```
skip_code
    This disables compiling code where not needed. We still compile tools,
    and code might be compiled as a consequence of other processing.

skip_export_make
    This disables compiling while the export version is constructed. This
    is mostly useful for debugging. Normally, one would want this test if
    the code was really being exported.

clean
    This requests a "make clean" before "make". This is reverse of the
    "opt out" convention for rebuilding makefiles, and perhaps should be
    changed to skip_clean? FIXME?

skip_no_libs
    This disables compiling of libraries as if most external libraries are
    not installed. By default, this script checks that library code is
    properly protected from this. An important exception is boost. Even if
    we pretend that most libs are not available, we still use boost. To
    check that we are robust to even boost being missing, use
    test_no_boost.

test_no_boost
    Provided that we are not skipping testing without libraries, we will
    still assume that boost is available. This option tests whether we can
    compile even if boost is not available.

skip_prog_no_libs
    This disables compiling of programs, including ones found in test
    directories, as if most external libraries are not installed. As with
    skip_no_libs, boost is excepted.

test_prog_no_boost
    In analogy with test_no_boost, this option test whether we can compile
    even if boost is not available in the case of programs.

skip_cxx
    We used to check that C code also compiles with the C++ compiler,
    and this option used to disable doing so. But now we do not do this
    unless requested by the option do_cxx, and skip_cxx does nothing
    (unless there is a preceding do_cxx in the options, in which case this
    would reset the default). We keep skip_cxx to avoid breaking old
    scripts/Makefiles, but we will remove this option at some point. Note
    that C++ code is always compiled with the C+ compiler and this
```

```
          option never disabled compiling of C++ code.

      do_cxx
          We used to check that C code also compiles with the C++ compiler.
          However, this is deprecated in clang, and also we suspect issuses with
          the linker with some versions of g+ on linux. Hence, we now do not
          default to doing so. This options enables this check. Note that C++
          code is always compiled with the C+ compiler, and this option does
          not request compiling of C++ code.

      skip_clean_src
          For export, source code is cleaned up to remove code protected by a
          number of #define's. This disables this behaviour, which can speed up
          debugging.

      skip_sub_dirs
          Skip library sub dirs. Note that if the directory is a dependency of a
          program being processes, it would get compiled anyway.

      skip_cpp_dirs
          Skip library *_cpp directories when src/lib is processed. Note that if
          the directory is a dependency of a program being processes, it would
          get compiled anyway.

      skip_test_dirs
          This option requests skipping test dirs.

      skip_test_sub_dirs
          This option requests skipping test sub dirs such as the common sub
          directory "interactive".

      skip_prog_dirs
          This option request skipping program dirs. This does not apply to
          library test directories.

NON_OPTION ARGUMENTS

      [ PROGRAM_DIR ]
           Any option that is not one of the above is tested as to whether it is
           a subdirectory of ${IVI_SRC_PATH}, which is usually ${HOME}/src. If
           this test succeeds, then PROGRAM_DIR is added to the list of
           specified directories to be converted into a distribution. If there
           is no PROGRAM_DIR argument, then we assume we are creating a
           distribution for a hard coded subset of programs, which we often use
           to check that things build.
```

```
EXAMPLES (MORE TO COME)
    To build a standalone version of a single program (e.g., hdp_hmm_lt),
    specified relative to the source directory (which holds Make, lib, and
    include_after) without extra checking of library code, assuming we are in
    the src direction and that we are using bash:

       Make/scripts/make_ivi export_only projects/hdp_hmm_lt \
           1>make_ivi.stdout 2>make_ivi.stderr &

AUTHOR
    Kobus Barnard
```

B.1.18 stream_file_names

```
stream_file_names(1)                                    stream_file_names(1)

IDENTIFICATION
    Lists names of files and dirs corresponding to various criteria.

DESCRIPTION
    This script is a collection of hacks to produce a sequence of lines of file
    or directory names that satisfy various conditions. Various flavors of
    various unix tools that one might use for this have trouble if the answer is
    empty. This script will simply output nothing if there is no match.

ARGUMENTS
    If an argument is given that is not identified as an option, then it is
    assumed to be a directory name to perform the action. Otherwise the current
    directory is used.

    If the option "-t" is given (for "test") then the script outputs "1" if
    there is a match, and nothing if there is no match.

    If the option "-v" is given, then the script outputs all files that do not
    match certain conditions after the files have been initially gathered under
    all options other than "-m", "-c_lib_dir", and -"cxx_lib_dir". For example,
    "-m -v" gives non-main files after the type has been specified. The behavior
    in more complex situations needs to be defined/studied further.

    The option "-no_path" asks for filenames only.
```

```
    If the option "-l" is given, then the file contents are streamed.

CURRENT TESTS (MOST ARE EXCLUSIVE, WHICH IS A BIT OBSCURE)
    -m          Files with a main() module. If source type is not given,
                    then -s is implied.
    -c          C files
    -x          Cpp files
    -s          Source files (C or Cpp)
    -h          Header files
    -a          All files
    -c2man      Files that we want to give to c2man
    -doxygen    Files that we want to give to doxygen
    -all_ivi    Dirs that we want for IVI or IVI_cpp
    -c_lib_dir  C library directories
    -cxx_lib_dir Cpp library directories
    -skip_incl  Skip files with names of the form *_incl.h
    -skip_all   Skip files with names of the form ivi_all_*.h
    -skip_export Skip files with names of the form ivi_export_*.h

AUTHOR
    Kobus Barnard
```

B.1.19 update_ivi_core

```
update_ivi_core(1)                                      update_ivi_core(1)

IDENTIFICATION
    A script to update core IVI software

DESCRIPTION
    This script updates the core IVI software using SVN. The "core" system is
    the build system, the library, some utility programs ("tools"), and some
    example programs. It currently excludes programs for research projects, but
    this might change. Alternatively, we might write a script "update_ivi_all".

    This script begins by finding the top of the src dir to update. If first
    consults the environment variable FORCE_IVI_SRC_DIR. If that is not set, it
    then checks if it is being run in $HOME. Since the IVILAB software system
    (IVISS) assumes a src directory that is not $HOME, it assumes that it should
    be updating the src directory that holds this script (typically, that is
    $HOME/src), if it looks OK. On the other hand, if the script is not being
    run in $HOME, it will check if the current directory is part of an IVISS src
    directory. If so, it will use that. Otherwise it will assume it is in a new
```

```
    (planned) src directory.

    Note that typically you get this script by getting the src/Make directory,
    which you also update by this script. So there is an initial boot-strapping
    process needed. One way to do this is to manually create the src directory,
    change to that, and then do a:
        svn co svn+ssh://vision.cs.arizona.edu/home/svn/src/Make/trunk Make
    followed by
        Make/scripts/update_ivi_core
    An alternative is to get a copy of update_ivi_core, say in your home
    directory, and do:
        export FORCE_IVI_SRC_DIR=[src_dir_to_create]
        ./update_ivi_core
    From this point on, there is no need to set FORCE_IVI_SRC_DIR. If you are in
    that directory, then you should be able to update using:
        Make/scripts/update_ivi_core
    Even more convenient, and generally recommended, is to have
    ${IVI_SRC_PATH}Make/scripts in your PATH.

    If you have already checked out "Make" into your source directory, and you
    are in that directory, then this script can be run as:
        Make/scripts/update_ivi_core
    Alternatively, and probably ideally, some instance of "Make/scripts" would
    be in your path.

AUTHOR
    Kobus Barnard
```

B.2 Build scripts that mostly support the build system

B.2.1 build_file_list

```
build_file_list(1)                                        build_file_list(1)

IDENTIFICATION
    A script to maintain the file Include_lines/file_list

DESCRIPTION
    Script called by make to maintain the Include_lines/file_list. This tracks
    changes in the source collection. If source files are added, deleted, or
```

```
    renamed, then the file Include_lines/file_list gets updated which leads to a
    rebuild of the makefiles on the next call to make.

    The reason why we need an additional call to make is that this script gets
    called whenever the directory gets updated which happens relatively
    freqently due to a number of reasons. However, this only indicates that the
    source file collection might have changed, and, in fact, source file
    collection changes are relatively rare. Thus there are many false alarms
    which we can prune relatively quickly. But having the trigger (directory
    timesamp) be a dependency in make will mean that we rebuild the makefiles
    every time which is relatively expensive. Hence we use the "hidden"
    dependency Include_lines/file_list. The script build-2, which is called by
    Makefile, manages this by calling make several times. If build-2 is not used
    (e.g., "make -f Makfeile-2) then it may take multiple makes before
    everything is up to date. Build-2 uses multiple calls to avoid this
    confusion.

    This script is not generally used in makefiles as we now usually call
    build_file_list_2 via ivi_lock_for_make directly in the Makefiles. However,
    the script build-2 uses it as the target is less clear there, and the build
    is expected not to be parallel.

AUTHOR
    Kobus Barnard
```

B.2.2 build_incl_dot_h

```
build_incl_dot_h(1)                                        build_incl_dot_h(1)

IDENTIFICATION
    A script to maintain the file <sub_lib_name>_incl.h

DESCRIPTION
    This script maintains the files *_incl.h in the directory provided as an
    argument. If the file <sub_lib_name>_incl.h does not exist, the script
    builds that file, and colatrally it also builds
    Include_lines/Makefile-depend-incl-dot-h.

    If the file <sub_lib_name>_incl.h already exists, this script simply touches
    it. Hence it is up to the make process to remove <sub_lib_name>_incl.h if it
    is out of date to force this script to rebuild it.

    We use the above mechanism because <sub_lib_name>_incl.h plays two roles in
```

```
    the build systems. First, it transfers the dependencies of the dot h files
    in the sub lib directory to targets depending on <sub_lib_name>_incl.h.
    Second, it must track whether it is up to date. We could
    simplyMakefile-depend-incl-dot-h force a rebuild in both cases, but the
    second case occurs much more often, and the forced rebuild would be a bit
    confusing and takes a bit of time (not much).

AUTHOR
    Kobus Barnard
```

B.2.3 build_include_lines

```
build_include_lines(1)                                  build_include_lines(1)

IDENTIFICATION
    A script to maintain the file Include_lines/include_lines.

DESCRIPTION
    This script is called by make to maintain the file
    Include_lines/include_lines. This file has two roles for efficiency. 1) Its
    existence/timestamp indicates whether the dependencies (i.e.,
    Makefile-depend) are up to date. 2) Its contents tracks the sub libs
    mentioned in include lines, and when this changes, it triggers the build of
    Makefile-dirs.

    The logic for (1) is implemented in the Makefiles, the script
    build_include_line_files, and the script update_include_line_file. By the
    time this script is called, if Include_lines/include_lines was out of date,
    it will have been removed, and it is up to this script to rebuild it.
    Regardless, this script then checks if the contents indicate that the
    sub-directories need to be updated, and calls other scripts to do that if
    needed. It is important that we do this only when needed, because sorting
    out the sub-libs is potentially expensive because it recurses on all
    sub-libs to look for their sub-libs.

    This script is really just a front end to build_include_lines_2 which does
    all the heavy lifting.

PARAMETERS
    This script takes either zero or one or two args. If the first arg is "-f",
    then we force the building of sub_libs_needed. If there is another argument,
    it is taken as the directory to run it in.
```

```
AUTHOR
    Kobus Barnard
```

B.2.4 build_makefile_libs_needed

```
build_makefile_libs_needed(1)                    build_makefile_libs_needed(1)

IDENTIFICATION
    A script to build the file Makefile-libs-needed

DESCRIPTION
    This script builds the file Makefile-libs-needed (technically,
    Include_lines/Makefile-libs-needed/Makefile-libs-needed${VAR_CACHE_SUFFIX})
    in the directory given by the first parameter, or the current directory if
    there are no parameters.

    This script might be superflous as it used to setup locking for
    build_makefile_libs_needed_2, but now we do it outside.

AUTHOR
    Kobus Barnard
```

B.2.5 finish_executable

```
finish_executable(1)                                    finish_executable(1)

IDENTIFICATION
    A script to permit execution and link executables and libraries to the
    current directory

USAGE
    finish_executable {exec_path} {exec_name}

DESCRIPTION
    This script ensures exectubles are permitted for execution, and links them
    to the current directory under the control of user setaable environment
    variables. It is used by the build system, and likely has no other use. We
    overload it so that it does both program executables and libraries.
    Libraries are also permitted executable as a side effect, which we could
    disable, but one can imagine shared libraries needed this on some systems (I
    have not checked this).
```

```
    We do this linking process because we want to maintain multiple instances of
    objects (e.g., dev and prodcution). It is possible to manage this in a
    cleaner way by changing your path when changing the build parameter, which
    is what Kobus does via scripts. However, folks newer to the system may
    wonder where the exectuble that was just built lives, as the path to it is a
    bit obscure. Hence our default is to make the links.

    The following environment variables are consulted in order when the are set
    to 1. Setting them to anything other than 1 will typically have the same
    effect, as the test is merely against being set, but is not advised for bash
    based shells, as the difference between not being set, and set to NULL, is
    slight in that case and is bound to lead to problems.
        DONT_LINK_EXECUTABLE
        FORCE_LINK_EXECUTABLE
        FORCE_COPY_EXECUTABLE
    The third variable is mostly used for shared objects, and is the default for
    "make py_mod"

AUTHOR
    Kobus Barnard
```

B.2.6 update_include_line_file

```
update_include_line_file(1)                    update_include_line_file(1)

IDENTIFICATION
    Updates files in Include lines tracking include lines of the a file

DESCRIPTION
    This script is called by make to update files in Include_lines that track
    the include lines for the file provided as an argument. If the process
    reveals a change from the previous state, then the makefile dependencies
    need to be rebuilt. To have ths happen, this script removes
    Include_lines/include_lines. It assumes that that the makefile will provide
    the needed call to build_include_lines to rebuild it and perform additional
    tasks if needed based on its new contents. Note that the script
    build_include_line_files does a similar thing over the whole directory.

    The include line tracking is achieved using two shadow files for each source
    file, ${file}, that track whether the include lines have changes for the
    source file. The shadow file Include_lines/new/${file} is updated whenever
    the source file changes. The change is propogated to the file
```

```
    Include_lines/${file} only if the include file list has changed.

    The script first creates/overwrites the include line file for the provided
    file in Include_lines/new. If this is different from the one in
    Include_lines, the later is updated, and the file
    Include_lines/include_lines is removed.

PARAMETERS
    This script takes either two or three arguments. If the first arg is "-f",
    then we force the building of sub_libs_needed. The first arg not "-f" is the
    file whose include files need to be updated. If there is another argument,
    it is taken as the directory to run it in.

AUTHOR
    Kobus Barnard
```