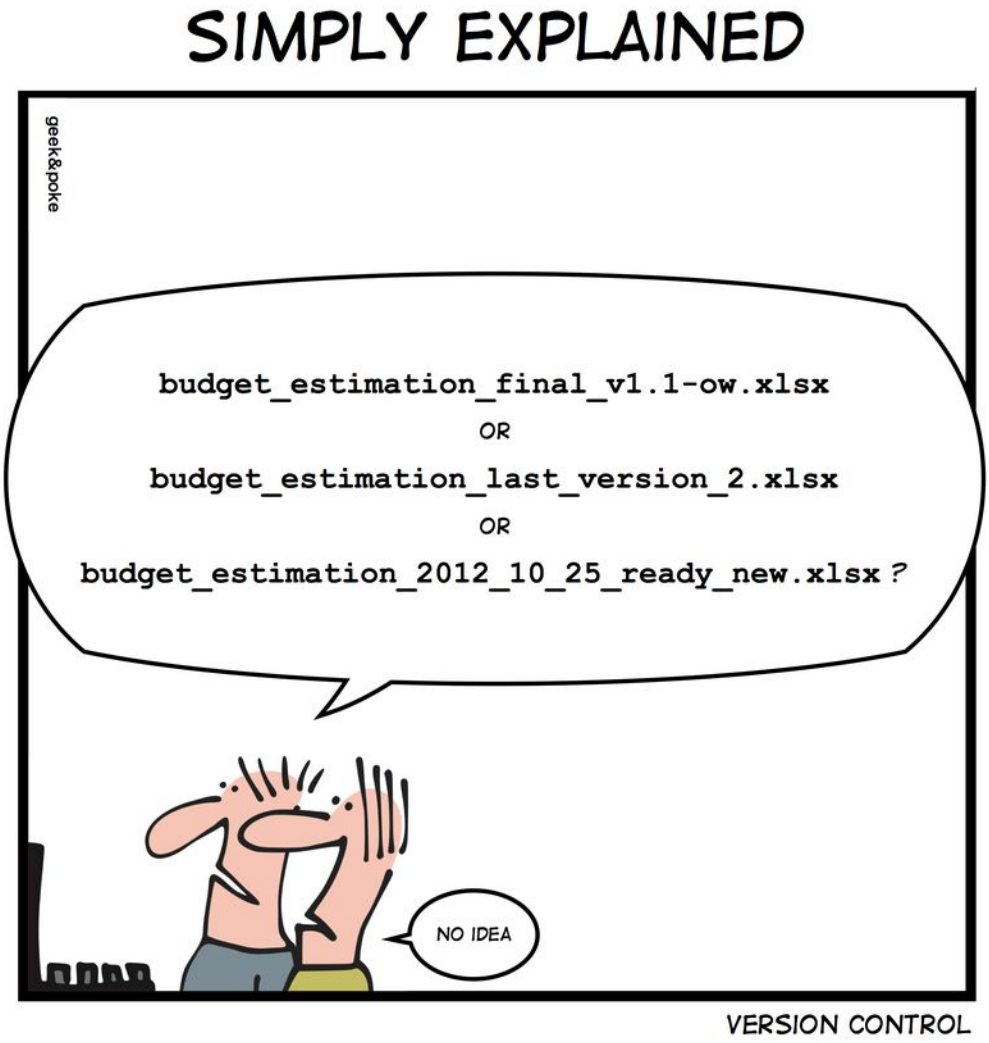
Project Code Management

Repositories, Documentation and Unittests



Frank Everdij, December 1, 2016

Table of Contents

Introduction

This document is meant for scientific personnel who are developing code within a research project environment.

It explains the benefits and use of a repository system such as Git or Mercurial and introduces a simple and powerful document system called Sphinx, which can combine various sources into different output formats suitable for publishing either as a web-page or as hard-copy.

By developing a standard layout for a project directory, the code source, documentation, data generation and article or poster directories can be separated. This results in easier referencing when creating documentation and a quicker comprehension and sharing of projects, which is important when working with colleagues and supervisors.

Moreover, automated procedures like build-tools and unit-testing can be applied to every standard project, making it easier to build and run code and spot problems in code or in external libraries.

Layout

First, the revision management system or repository will be introduced, since it is central for the use of code and documentation in a project directory. We will cover its installation and give examples of local and remote usage.

Next, the Sphinx document system will be explained. Its installation requirements and syntax will be reviewed in short.

Then a Python tool will be presented to perform unit-tests on data produced from programs. This tool can perform other tasks, such as building the program and preparing documentation.

Lastly, the project directory structure will be explained. It will be shown how the repository system should be used in combination with the project.

Windows support

Although the document is written for people using Linux systems with a basic development environment like the GNU toolchain and editors, the knowledge can be applied to windows systems with Visual Studio as development environment.

People using windows systems do need to separately install the tools to use a repository and documentation system: Some links for downloading software have been supplied in the Links chapter. Please choose the most recent stable program versions and install the `x86_64` (64-bit) program versions where possible.

Repository

Revision management

A revision management system allows for consistent incremental development of a directory tree by one person or multiple people. It tracks file changes by authors, does version labeling, creates differential files between versions, and allows for forks or branches of the code coexisting side-by-side.

It it mostly used for managing program code written in a modern programming language like C++ or Python, but it can also be used for documentation, either in text or in a markup/formatting language like LaTeX, HTML, RST or XML.

In this document we will focus on code management of software developed in C or C++ language.

The main point of having a revision management system is the ability to make a snapshot of the code at certain development events, like implementing a new feature, adding a new function or even changing numerical constants in the code. By making this snapshot and adding a comment why this snapshot was taken, you document and isolate the changes you have made from the previous version. This is essential in finding code bugs, regardless whether they are caused by your edits to the code or someone elses.

To make optimal use of the revision management system, you need to treat the system as an additional layer on top of the general accepted guidelines for programming code: Of course you should write clean code by proper indentation and formatting, writing code-comments, create headers for functions and subroutines, and provide makefiles and short instructions how to compile and use it.

But it is not a cure or a fix for writing bad code or bad programming. It does reveal however when and where a specific change has occurred that broke the program by allowing you to travel back and forth between snapshots of the code and test its functionality.

Choice

Mercurial

Written mostly in Python, with a small C library needed for binary file diff functionality, its aim is to be an easily adoptable revision system. This is evident from its inheritance of the Python philosophy that each problem has one preferred solution: In general there is one preferred way of performing a certain action in Mercurial. It has about 30 commands with some having additional arguments.

Mercurial holds on to removed files in favor of preserving the revision history list, but this also means that errors and removed files will remain present in the entire repository and that there is no easy way to remove this ‘dead weight’ which people cite as a disadvantage. It is also considered a bit ‘boring’ to use since there are few commands and there isn’t much to be learned by advanced Mercurial users.

Git

Git was created as a tool to merge code contributions into the Linux kernel system, supervised by Linus Torvalds himself. Forced to drop the commercial versioning tool BitKeeper and pressed for time, he quickly produced Git in order to use it to merge changes for the linux kernel citing performance and non-linear code-development as main features. Since then it has been constantly maintained and developed and made available online as github.com as a service to store open source code projects.

Its advantages is that it is fast, powerful and flexible, allowing actions on a ‘deep’ repository level, for instance changing history commits. Also it is the de-facto standard for large distributed open source projects in combination with github.com with the linux kernel as a famous example.

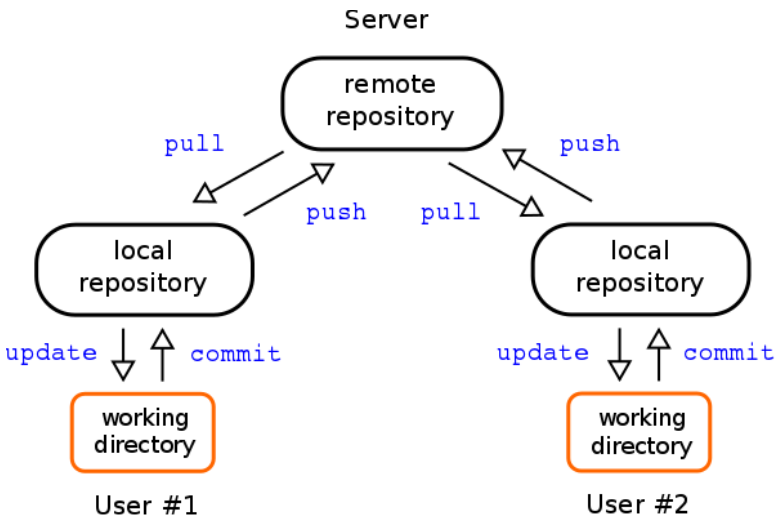
An often mentioned disadvantage is the variety of executable programs written in C or Ruby and scripts (in Bash or Perl) which exist to extent the git functionality. Consequently there are many less-known git commands and scripts to perform specific or particular tasks. Another problem with the choice of git executables was that windows support lagged behind the Unix git-releases, although there is a working up-to-date command-line version of git these days.

Mercurial has become our preferred choice in code management because of its simplicity and robustness and the relative small size of our code projects. Added to that we got mercurial-server which is a distributed version of mercurial: hosted on a private server and access granted by ssh public key, it acts as a safe remote repository accessible from anywhere.

Distributed repository

Mercurial (and Git) is an implementation of a distributed revision system. This means that there can be a central repository where revisions are being stored, but creating those revisions is being performed on local copies of that repository.

The diagram below shows the dataflow between a mercurial repository located on a remote server and local copies on users' laptops or workstations.



Users can work independently by committing changes to their local repository, but can also share their code-changes by pushing this local repository to the server, where it can be pulled and updated by another user.

This can cause conflicts if a file is edited by more that two users and is pushed to the central repository. Mercurial detects these conflicts and will offer tools to resolve this, e.g. with `hg merge`.

There are many advantages of a distributed repository over older central repositories (like Subversion):

1) The distributed repository only acts as an exchange-hub for code revisions, it can be the 'central' repository, but doesn't have to be.
2) Contrary to pure centralized repository systems there is no bottleneck in checking code in or out due to server load or congestion, since the entire history is stored locally.
3) … which also means the speed of revision management is much faster since apart from pushing or pulling everything is processed on the local disk.
4) The local copies are full copies of the central repository, so if for whatever reason the central repository is unavailable or even destroyed, users can remain working on their copy and a working central repository can be recreated from local copies.

Installation

You can install Mercurial in Debian/Ubuntu systems by issuing:

```
$ sudo apt install mercurial kdiff3 tortoisehg
```

For older versions of Ubuntu systems, you can use `apt-get` instead of `apt` .

`kdiff3` is a program which helps you resolve conflicts when performing a merge operation with `hg merge` . When merging a file between two revisions and conflicts arise because each revision has different content of a specific line, `kdiff3` shows you the differences and lets you resolve the conflict by choosing which line takes preference and editing the result. Merging will be addressed in detail later when dealing with remote repositories.

`tortoisehg` is a graphical user interface which can display the Mercurial revision tree graphically and lets you browse the revision database for commit logs and diffs. You can run it by entering:

```
$ thg
```

Before using Mercurial it is wise to set a few defaults in `~/.hgrc` , the settings file in the user's home directory. This is my default:

```
# User Interface
[ui]

# show changed files and be a bit more verbose if True
verbose = true

# username data to appear in commits: "Joe User <joe.user at host.com>"
username = Frank Everdij <F.P.X.Everdij@tudelft.nl>

# program to use as editor (vi is the default editor)
editor = gedit -w --new-window

# merge program
merge = kdiff3

[merge-tools]
kdiff3.args = $base $local $other -o $output
```

Starting a local repository

I will illustrate the use of mercurial by creating a repository from an existing code directory example. The example directory is called `helloworld` and contains `helloworld.c` and a `Makefile` :

```
~/helloworld$ ls
helloworld.c  Makefile
~/helloworld$ cat helloworld.c
#include <stdio.h>

int main (void)
{
     printf("Hello World!\n");
     return 0;
}
```

Initialization

To start a new repository for the current directory, enter `hg init` :

```
~/helloworld$ hg init
```

This will create the `.hg` folder and initializes the repository to its default values:

```
~/helloworld$ ls -a
.  ..  helloworld.c  .hg  Makefile
```

At this stage nothing is in the repository yet. You can check this by entering `hg log` :

```
~/helloworld$ hg log
~/helloworld$
```

Adding files

To add files, use `hg add <filename>` to add the file `<filename>` to the repository. Mercurial will then display a confirmation that it has added the file to the repository. If you want to add all files, just use `hg add` :

```
~/helloworld$ hg add
adding Makefile
adding helloworld.c
```

...which adds every single file in that directory and subdirectories into the repository.

This can cause problems when the files you add are not really meant to be in the repository, like object files or zipped data files.

To prevent these files from being added, you can put the filenames or a matching glob pattern in a special file called `.hgignore` in the directory root. Mercurial will check this file every time you perform a `hg add` and will ignore files matching the name or patterns.

A simple example of such an `.hgignore` file is:

```
syntax: glob
*~
*.o
*.a
*.zip
*.gz
*.bz2
helloworld
```

Which tells Mercurial to ignore gedit backup files, object files, archive files, and the most common compressed file extensions as well as the executable `helloworld` , since it can be generated from the source file.

You can check the repository status with `hg status` :

```
~/helloworld$ hg status
A Makefile
A helloworld.c
? .hgignore
```

This will tell you what files are (`A`)dded and what files are still unknown (`?`) to the Mercurial repository. You can either add the unknown files or use `hg forget <filename>` to tell Mercurial to ignore these files. In our case we forgot to include `.hgignore` so we add it explicitely :

```
~/helloworld$ hg add .hgignore
~/helloworld$ hg status
A .hgignore
A Makefile
A helloworld.c
```

Testing and committing

Before creating a new revision we want to make sure that the program builds and works. Also, when building and running a program, extra files may be created and we have to check whether these extra files are to be included or should be ignored. So `make` the program and run it :

```
~/helloworld$ make
gcc -O2 -g -Wall -c helloworld.c -o helloworld.o
gcc -o helloworld helloworld.o
~/helloworld$ ./helloworld
Hello World!
```

It works. Now check if there were extra files being created :

```
helloworld$ ls
helloworld  helloworld.c  helloworld.o  Makefile
```

There are two extra files created, `helloworld.o` and `helloworld`, which we do not want to add to the repository, so the question is: Will they be ignored by Mercurial as specified in `.hgignore` ? :

```
~/helloworld$ hg status
A .hgignore
A Makefile
A helloworld.c
```

They are indeed being ignored, so `.hgignore` works.

After this compile-and-run check, we are ready to create a new revision by performing a `hg commit` .This creates a new revision in the repository with the files you just added. In order to give a description of this revision, `hg commit` will start a text editor (`/bin/nano` by default) to enter a brief explanation of this commit. The description is required: Leaving it blank will abort the commit.

Alternatively you can add a one-liner via the command line with `hg commit -m "your revision description here"` :

```
~/helloworld$ hg commit -m "first commit for helloworld C code program"
~/helloworld$ hg log
changeset:   0:04064931dd4c
tag:         tip
user:        Frank Everdij <F.P.X.Everdij@tudelft.nl>
date:        Fri Oct 28 14:33:48 2016 +0200
summary:     first commit for helloworld C code program
```

This completes a new revision or ‘changeset’ of your repository, as the above `hg log` confirms.

You can now continue adding or editing code files, test your code and make changes until you have reached a point where you want to make a new revision. Simply running `hg status` again will tell you which files are modified. Performing an `hg commit` again will save the changes into the repository.

A very short example of re-editing and re-committing changes follows next:

I have edited the `"Hello world!"` string from the previous example into the Dutch version.

`hg status` mentions the change and a `hg diff` shows exactly what has changed:

```
~/helloworld$ hg status
M helloworld.c
~/helloworld$ hg diff
diff -r 04064931dd4c helloworld.c
--- a/helloworld.c    Fri Oct 28 14:33:48 2016 +0200
+++ b/helloworld.c    Fri Oct 28 14:44:00 2016 +0200
@@ -2,6 +2,6 @@

 int main (void)
 {
-     printf("Hello World!\n");
+     printf("Hallo Wereld!\n");
      return 0;
 }
```

After making and running the program it produces:

```
~/helloworld$ make
gcc -O2 -g -Wall -c helloworld.c -o helloworld.o
gcc -o helloworld helloworld.o
~/helloworld$ ./helloworld
Hallo Wereld!
```

I am happy with the result so I commit it:

```
~/helloworld$ hg commit -m "changed string into dutch"
~/helloworld$ hg log
changeset:   1:9d437e267a79
tag:         tip
user:        Frank Everdij <F.P.X.Everdij@tudelft.nl>
date:        Tue Nov 15 11:57:43 2016 +0100
summary:     changed string into dutch

changeset:   0:04064931dd4c
user:        Frank Everdij <F.P.X.Everdij@tudelft.nl>
date:        Fri Oct 28 14:33:48 2016 +0200
summary:     first commit for helloworld C code program
```

A new revision or ‘changeset’ 1 appeared which has become the current revision or ‘tip’ of the repository. This is because I have made changes to the ‘old’ changeset 0 and the changes were committed, thus creating a new revision.

Revision history and update

Because Mercurial keeps track of the changes you perform to code or text, would it be possible to go back to a previous revision and run the program in its original revision 0 ?

Yes: This is possible with the `hg update` command. But first make sure that the directory does not have uncommitted changes, since you would lose those changes by going to a previous revision. Check this with `hg status`:

```
~/helloworld$ hg status
~/helloworld$
```

No changes are reported. Now you can update the code directory with a previous revision:

```
~/helloworld$ hg update 0
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

Let's re-make the program and run it again:

```
~/helloworld$ make
gcc -O2 -g -Wall -c helloworld.c -o helloworld.o
gcc -o helloworld helloworld.o
~/helloworld$ ./helloworld
Hello World!
```

Correct, this was the first revision 0 of the code directory. A `grep` of the code confirms the original statement being present:

```
~/helloworld$ grep printf helloworld.c
     printf("Hello World!\n");
```

You can go back to the current revision by typing `hg update`. Its default argument is the latest revision:

```
~/helloworld$ hg update
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
~/helloworld$ make
gcc -O2 -g -Wall -c helloworld.c -o helloworld.o
gcc -o helloworld helloworld.o
~/helloworld$ ./helloworld
Hallo Wereld!
```

And the current revision is restored.

This concludes an example of the use of Mercurial for a code directory on local storage media.

Remote repository

Performing revision control on a local directory has the disadvantage that only you can work on that code in that directory and when disaster strikes and your disk crashes, you lose your work if you don't have a backup.

To fix problems like these and more importantly providing the group with a method of storing and archiving code on a secure private server, I have installed mercurial-server on our cluster to act as our main remote repository for codes and projects.

To access these remote repositories on the cluster, it is important to have secure shell configured in such a way that you have made a public/private keypair and set up agent forwarding. For details, see Appendix A.

Cloning

For editing a remote repository, you need to have a local copy of the repository. Assuming you have performed the necessary steps with setting up secure access, the command to copy a repository is:

```
$ hg clone ssh://hg@sissy.tudelft.net/feverdij/example example
```

This command 'clones' the remote '`example`' repository to the '`example`' directory on your local drive. Vice versa, you can create remote repositories by cloning to it, i.e. reversing the arguments of the clone command:

```
$ hg clone example ssh://hg@sissy.tudelft.net/feverdij/example
```

This works if the local '`example`' directory contains a Mercurial repository database created via `hg init`.

Permissions

An additional requirement for creating a remote repository is that the person associated by the public key has '`init`' permission by the mercurial-server. This permission controls whether people are allowed to create a repository on the server from scratch.

Currently this permission is not given to PhD's and students, only to supervisors and administrators. However once created, PhD's and students have '`write`' permission to a repository when the project is active and the code is in development.

For archived projects, '`read`' permissions are sufficient for users to download the repository in order to read documentation, study the contents and run the examples. You can still edit code and continue to commit changes locally, but for those users 'pushing' changes to the remote repository is not possible.

Pull and Push

Once a repository is available remotely and you have cloned it to your local drive (or you cloned a local repository to a remote server) you can make changes to it and commit them. But how do you upload changes to the remote repository or download updates from it?

This is where the mercurial commands `hg pull` and `hg push` do their work.

`hg pull` checks if there are updates to the remote repository and downloads or 'pulls' the changes to the local repository. Its syntax is easy:

```
$ hg pull ssh://hg@sissy.tudelft.net/feverdij/example
```

To prevent typing the remote path every time you want to perform a pull (or a push) you can set the default path in the `.hg/hgrc` file:

```
# path entries
[paths]
default = ssh://hg@sissy.tudelft.net/feverdij/example
```

So the next time it is sufficient to type:

```
$ hg pull
```

`hg push` is the opposite of `hg pull`. It compares the local and remote repositories and if the local repository has new additions, `hg push` will upload or 'push' those additions to the remote repository:

```
$ hg push ssh://hg@sissy.tudelft.net/feverdij/example
```

or

```
$ hg push
```

if you have added a default entry in `.hg/hgrc`

`hg push` is different from `hg pull` in handling merge conflicts: if `hg push` finds a new revision (or 'branch') in the remote repository, it will abort and tell you to pull and merge that new revision, then push it again.

Update

The pulled repository is not yet synchronized with the files in the directory. Similar to the local repository example of switching revisions, you must update the files with the information from the new repository:

```
$ hg update
```

This will update all the files which are tracked by Mercurial to the latest revision.

Example

Let us show you an example what working with a remote repository looks like. We will clone a repository, in this case a jemjive C++ code repository, make changes to a file, committing that change and uploading or ‘pushing’ the new revision.

First perform the clone:

```
~$ hg clone ssh://hg@localhost:22022/feverdij/umfpack umfpack
requesting all changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 5 changes to 5 files
updating to branch default
5 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

Listing the files in the repository gives:

```
~$ cd umfpack
~/umfpack$ ls
Makefile  poisson.cpp  UmfpackSolver.cpp  UmfpackSolver.h
```

Looking at the log, it tells us that this is the first and only revision:

```
~/umfpack$ hg log
changeset:   0:c363445b38d4
tag:         tip
user:        Frank Everdij <f.p.x.everdij@tudelft.nl>
date:        Tue Nov 15 19:16:27 2016 +0100
summary:     FE: small example of UMFPACK solver in poisson equation
solve
```

The UmfpackSolver code file has a bug which passes an incorrect matrix structure to external solver libraries. This needs to be fixed:

```
~/umfpack~$ vi UmfpackSolver.cpp
~/umfpack~$ hg diff
diff -r c363445b38d4 UmfpackSolver.cpp
--- a/UmfpackSolver.cpp    Tue Nov 15 19:16:27 2016 +0100
+++ b/UmfpackSolver.cpp    Tue Nov 15 20:10:58 2016 +0100
@@ -221,7 +221,7 @@
   {
     // Compute the new soluton and residual vector.

-    umferr = umfpack_dl_solve ( UMFPACK_A,
+    umferr = umfpack_dl_solve ( UMFPACK_At,
                                 offsets_.addr (),
                                 indices_.addr (),
                                 values_ .addr (),
```

It’s a simple fix, but let’s make sure the program works correctly by making and running it:

```
~/umfpack$ make
g++ -Wall -fstrict-aliasing -m64 -msse2 -mtune=native
-I/usr/local/include -I/home/frank/projects/jemjive/jive-2.2/include
-I/home/frank/projects/jemjive/jem-2.2/include -c -o OBJ/UmfpackSolver.o
UmfpackSolver.cpp
g++ -Wall -fstrict-aliasing -m64 -msse2 -mtune=native
-I/usr/local/include -I/home/frank/projects/jemjive/jive-2.2/include
-I/home/frank/projects/jemjive/jem-2.2/include -c -o OBJ/poisson.o
poisson.cpp
g++ -Wall -fstrict-aliasing -m64 -msse2 -mtune=native -rdynamic
-L/usr/local/lib -L/usr/local/atlas/lib
-L/home/frank/projects/jemjive/jive-2.2/lib
-L/home/frank/projects/jemjive/jem-2.2/lib -o example
OBJ/UmfpackSolver.o OBJ/poisson.o -lumfpack -lamd -lsuitesparseconfig
-llapack -lcblas -lf77blas -latlas -lgfortran -ljivembody -ljivefemodel
-ljivemesh -ljiveimplict -ljivesolver -ljivegl -ljivefem -ljivegraph
-ljivegeom -ljiveapp -ljivemodel -ljivealgebra -ljivemp -ljiveutil
-ljemgl -ljemxutil -ljemxml -ljemnumeric -ljemmp -ljem -lglut -lGLU -lGL
-lX11 -lmpi -lz -lreadline -lpthread -lm

~/umfpack$ ./example
Assembling the global matrix ...
Solving the system of equations ...
UMFPACK
Total run time is around :   5.16535  seconds
Writing the solution to `poisson.out' …
```

Re-checking the repository with `hg status`:

```
~/umfpack/$ hg status
M Makefile
M UmfpackSolver.cpp
```

I forgot that I also have changed the Makefile to compile this JemJive program on my system. I'll leave the change in the repository, since the changed Makefile will also work on other systems. Now perform the commit:

```
~/umfpack$ hg commit -m "FE: Fixed transpose bug in UmfpackSolver
driver"
```

...and push the new revision to the remote repository with `hg push`:

```
~/umfpack$ hg push
pushing to ssh://hg@localhost:22022/feverdij/umfpack
searching for changes
remote: adding changesets
remote: adding manifests
remote: adding file changes
remote: added 1 changesets with 2 changes to 2 files
remote: notify: sending 1 subscribers 1 changes
```

All done. Note the `remote: notify:` line. The mercurial-server package has been configured with a notify hook which sends emails to subscribed addresses when repositories are receiving a push. This is configurable by supervisors and administrators.

Merging

Sometimes a remote repository has changed when working locally on a clone of that repository. When you push your changes, Mercurial will detect a new revision on the remote side which conflicts with your revision.

In order to resolve such conflicts, Mercurial asks you to pull the remote repository and then attempts to automatically merge the differences between the revisions. But if there are conflicts within a file, manual merging is neccesary.

Manually merging files between different revisions is hard and sometimes isn't possible unless the file is modified. Which is why you should try to avoid such situations by communicating between repository members and making sure your local copy is up to date before working on it.

But sometimes you have to do a manual merge. I will illustrate this with a small local example. You can reproduce this example on your system to become familiar with merging, so you have an idea what to do if you encounter such a merge.

This is the example file, called `myfile` :

```
This is the first line.
En dit is de tweede regel.
The third line ends this file.
```

Create the repository `myrepo` and copy the file to that directory:

```
~$ hg init myrepo
~$ cd myrepo
~/myrepo$ cp /tmp/myfile .
~/myrepo$ ls -a
.  ..  .hg  myfile
```

Add the file:

```
~/myrepo$ hg add
adding myfile
```

And commit:

```
~/myrepo$ hg commit -m "first commit"
committing files:
myfile
committing manifest
committing changelog
committed changeset 0:1ad9cc1e4016
```

This is our initial repository. I will now make a clone `myclone` of this repository and edit `myfile` and commit it, but I will also do a different edit and commit of `myfile` on `myrepo`.

```
~/myrepo$ hg clone . ../myclone
updating to branch default
resolving manifests
getting myfile
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
~/myrepo$ cd ../myclone
~/myclone$ ls -a
.  ..  .hg  myfile
```

The clone is successful, now edit the file and commit. For `myclone` I've chosen to translate the second line back into English and rephrased the third line:

```
~/myclone$ vi myfile
~/myclone$ hg diff
diff -r 1ad9cc1e4016 myfile
--- a/myfile    Wed Nov 16 15:12:11 2016 +0100
+++ b/myfile    Wed Nov 16 15:14:25 2016 +0100
@@ -1,3 +1,3 @@
 This is the first line.
-En dit is de tweede regel.
-The third line ends this file.
+And this is the second line.
+The third line is where this file ends.
~/myclone$ hg commit -m "changed second line into English and rephrased
third line"
committing files:
myfile
committing manifest
committing changelog
committed changeset 1:9d1fdb4348d5
```

The head or 'tip' of `myclone` is now:

```
~/myclone$ hg tip
changeset:   1:9d1fdb4348d5
tag:         tip
user:        Frank Everdij <F.P.X.Everdij@tudelft.nl>
date:        Wed Nov 16 15:15:17 2016 +0100
files:       myfile
description:
changed second line into English and rephrased third line
```

For the original `myrepo`, i've added a fourth line to `myfile` and changed line three:

```
~/myrepo$ vi myfile
~/myrepo$ hg diff
diff -r 1ad9cc1e4016 myfile
--- a/myfile    Wed Nov 16 15:12:11 2016 +0100
+++ b/myfile    Wed Nov 16 15:17:51 2016 +0100
@@ -1,3 +1,4 @@
 This is the first line.
 En dit is de tweede regel.
-The third line ends this file.
+The third line does not end this file,
+but the fourth line will...
~/myrepo$ hg commit -m "changed third line and added fourth line"
committing files:
myfile
committing manifest
committing changelog
committed changeset 1:a091d77dc081
```

The tip of `myrepo` is now:

```
~/myrepo$ hg tip
changeset:   1:a091d77dc081
tag:         tip
user:        Frank Everdij <F.P.X.Everdij@tudelft.nl>
date:        Wed Nov 16 15:18:41 2016 +0100
files:       myfile
description:
changed third line and added fourth line
```

Notice that both tips have revision 1 as number. This and the fact that the content of `myfile` differs will cause conflicts when performing a `hg push` or `hg pull`. Going back to myclone and performing a `hg push` demonstrates this:

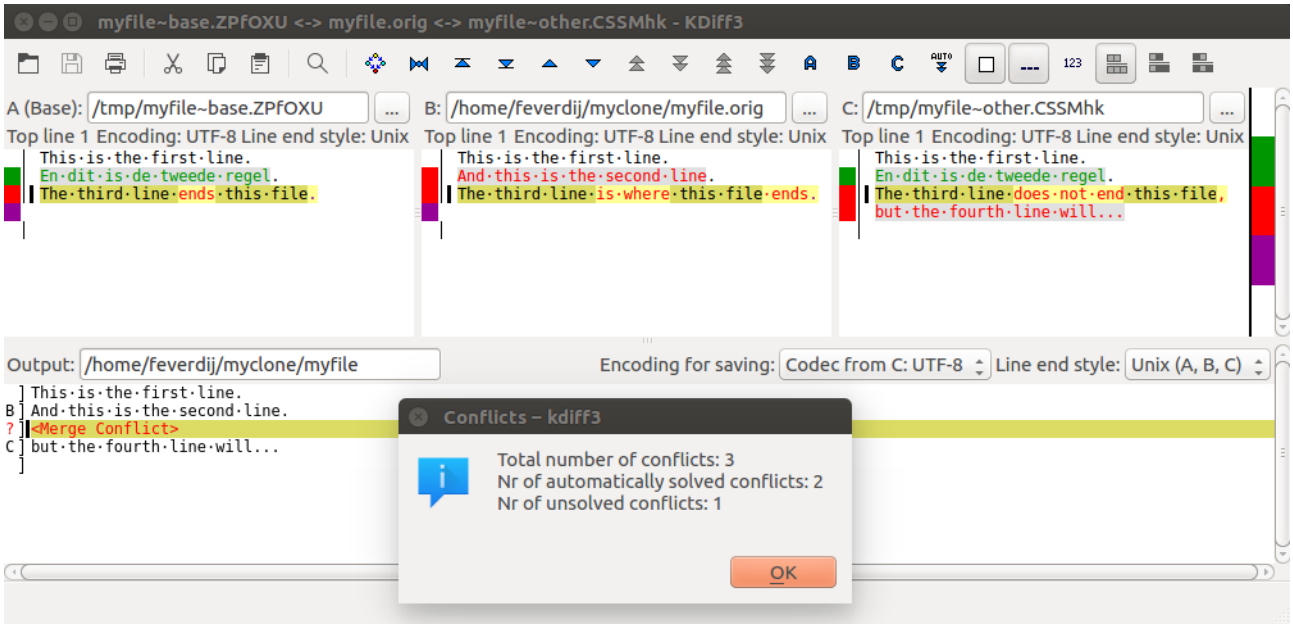
```
~/myclone$ hg push
pushing to /home/feverdij/myrepo
searching for changes
remote has heads on branch 'default' that are not known locally:
a091d77dc081
new remote heads on branch 'default':
 9d1fdb4348d5
abort: push creates new remote head 9d1fdb4348d5!
(pull and merge or see "hg help push" for details about pushing new
heads)
```

Mercurial aborted the `hg push` and suggests that we do a `hg pull` first, followed by `hg merge`. So let's do the `hg pull` first:

```
~/myclone$ hg pull
pulling from /home/feverdij/myrepo
searching for changes
1 changesets found
uncompressed size of bundle content:
     242 (changelog)
     168 (manifests)
     197  myfile
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files (+1 heads)
(run 'hg heads' to see heads, 'hg merge' to merge)
```

Mercurial now suggests `hg merge` to perform a manual merge, and we expect `kdiff3` to pop up:

```
~/myclone$ hg merge
```



There it is: `kdiff3` has launched, showing us four views of `myfile` and mentioning three conflicts of which one we need to resolve manually.

The top three views are marked A B and C and are respectively:

A) The original revision 0 `myfile` with three lines (both repositories have this)

B) The changed `myfile` from `myclone` which has the second line translated in English and a change in line three.

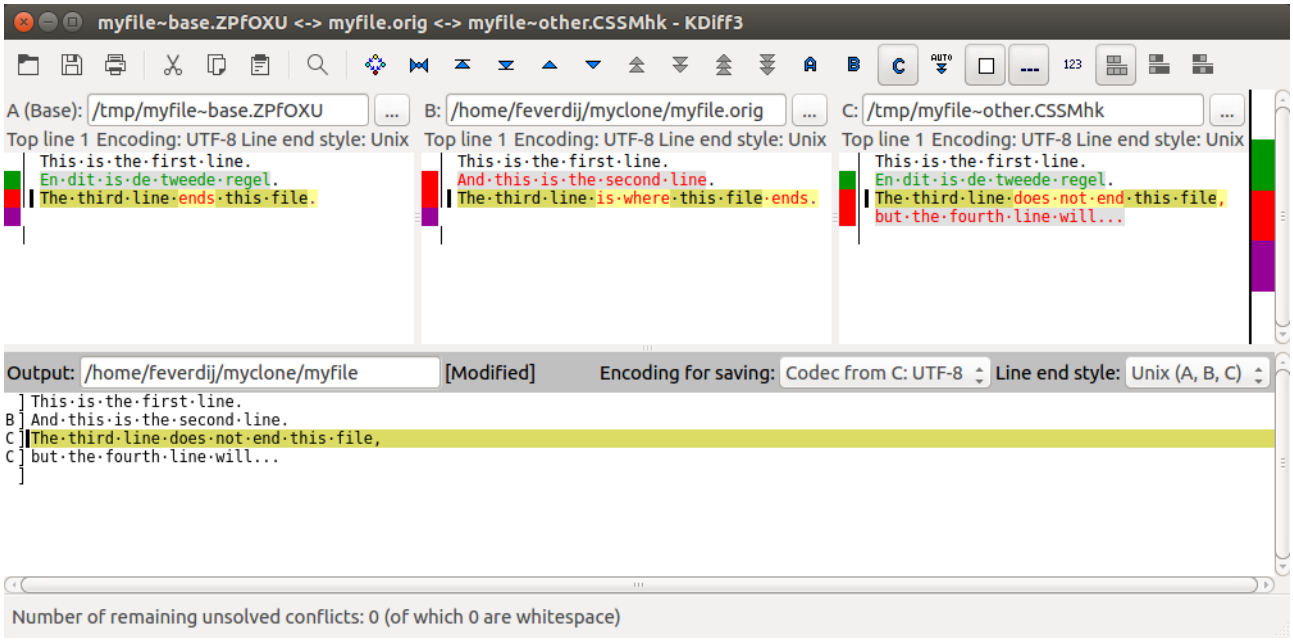
C) The changed `myfile` from `myrepo` which has the fourth lined added (and the third line changed)

The bottom view is the output view, where the result of the merge will be displayed. When you're done with the merge, this output will be saved to `myfile` in the `myclone` repository.

Mercurial has already solved two conflicts for us. The second line was chosen from `myclone` because it was edited in `myclone` but left untouched in `myrepo`. The fourth line was chosen from `myrepo` since the fourth line only exist there, and not in the earlier revision or in `myclone`.

The only thing we have to do is to resolve the conflict for the the third line. This is marked by the red question mark in the output view.

Selecting the [C] button in the `kdiff3` window copies the line from the C view (which is the most obvious choice to resolve the conflict) into the output window.



`kdiff3` reports no more conflicts, and by saving the result and quitting `kdiff3` `hg merge` finishes:

```
~/myclone$ hg merge
resolving manifests
merging myfile
0 files updated, 1 files merged, 0 files removed, 0 files unresolved
(branch merge, don't forget to commit)
```

We have changed `myfile` in `myclone` and resolved all conflicts but Mercurial warns us that the changes are not in the repository yet. We must perform a `hg commit` and it is usually customary to do this immediately after a merge with the comment "merge":

```
~/myclone$ hg commit -m "merge"
committing files:
myfile
committing manifest
committing changelog
committed changeset 3:28a966bc2ecf
```

if we look at the log we can see what happened:

```
~/myclone$ hg log
changeset:   3:28a966bc2ecf
tag:         tip
parent:      1:9d1fdb4348d5
parent:      2:a091d77dc081
user:        Frank Everdij <F.P.X.Everdij@tudelft.nl>
date:        Wed Nov 16 17:00:15 2016 +0100
files:       myfile
description:
merge

changeset:   2:a091d77dc081
parent:      0:1ad9cc1e4016
user:        Frank Everdij <F.P.X.Everdij@tudelft.nl>
date:        Wed Nov 16 15:18:41 2016 +0100
files:       myfile
description:
changed third line and added fourth line

changeset:   1:9d1fdb4348d5
user:        Frank Everdij <F.P.X.Everdij@tudelft.nl>
date:        Wed Nov 16 15:15:17 2016 +0100
files:       myfile
description:
changed second line into English and rephrased third line

changeset:   0:1ad9cc1e4016
user:        Frank Everdij <F.P.X.Everdij@tudelft.nl>
date:        Wed Nov 16 15:12:11 2016 +0100
files:       myfile
description:
first commit
```

The merge has created an intermediate revision 2 which adds the changes from `myrepo` and after the commit created revision 3 which completed the merge.

All that is left is a push to `myrepo` , the original repository:

```
~/myclone$ hg push
pushing to /home/feverdij/myrepo
searching for changes
2 changesets found
uncompressed size of bundle content:
     462 (changelog)
     332 (manifests)
     344  myfile
adding changesets
adding manifests
adding file changes
added 2 changesets with 2 changes to 1 files
```

in myrepo, run `hg update` to update `myfile` to the latest revision, but check first:

```
~/myrepo$ hg status --rev tip
M myfile
~/myrepo$ hg update
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

Now the two repositories are in sync and have been merged into one.

Tips

Do's:

Use `hg commit` very frequently ... and `hg status` just before a commit. Make sure that all files which are modified are saved and that all files are added.

Create a `.hgignore` file immediately after `hg init` . This will prevent inclusion of unnecessary files when doing an `hg add` . You can always edit `.hgignore` to allow files to be included later: It is better to add a minimal set of files to the repository in the beginning and expand later on, than to put too many files in the repository and do a `hg forget` or `hg remove` later.

Oh, and don't forget to add the `.hgignore` file to the repository with `hg add .hgignore`

Dont's:

Do not run `hg commit` in the middle of editing files! Usually, editors copy the file into a temporary file or buffer which Mercurial cannot see and any changes saved to a file after the commit is considered an extra change which you have to merge or commit again. You can introduce conflicts in the Mercurial database this way which you can recover from with `hg update --clean` , but chances are that you lost all your edits.

Do not include generated files from a Makefile, LaTeX or build script if the source files are already added and tracked in the Mercurial database: Examples are object, postscript, DVI, or PDF files and executables. Adding those generated files is just doubling Mercurial's work.

Do not include zipped or compressed files into a mercurial repository unless absolutely necessary: Mercurial will perform a binary diff on these files; zipped files in general do not have similar datablocks from previous revisions, so Mercurial has to store these in full.

Do not include security-sensitive or huge datafiles to the repository: Once added, it will track that file until it is removed by `hg remove` or `hg forget` . But once removed Mercurial will still hold the file in the repository: This is because it should be possible to go to a revision where the file is still being tracked.

Mercurial does not have a command of removing individual files completely from history, because anyone who has pulled a copy of the repository will have that file in their local repository: Therefore, a command to delete such a file from the database is useless.

Things to remember:

Always check if there are new revisions by doing a `hg pull` before resuming a project which is cloned from a remote repository. This saves you from the overhead of a possible big `hg merge` .

If you accidentally erase, trash or modify a file while editing or running a program and you want to undo that, use `hg revert <filename>` . The file will be backed up to `<filename>.orig` and Mercurial wil recreate that file from the most recent committed revision.

If you accidentally committed a change with a file missing or a large bug you just spotted and want to undo that commit immediately, use `hg rollback` . You can only rollback one commit!

Mercurial can track empty files, but not empty directories. To work around this issue, create a hidden file `.hidden` with zero length. You can use the command `touch` to do that. Then include that directory in the repository:

```
$ touch emptydir/.hidden
$ hg add emptydir/.hidden
```

How can you tell if the files in the directory which are tracked by Mercurial are the files from the most recent revision without doing `hg update`?

`hg status --rev tip` will tell you if files are different from the tip of the repository.

Documentation

Historically, documenting source code was done either in comment lines in the code or in an external file called `readme.txt`.

But for research projects the documentation should be much more comprehensive: It should explain what the code is trying to achieve and what algorithm it uses. It should explain the creation and inclusion of certain figures or even movies from the data produced by the code. Environmental parameters like operation system version, compiler version and the PC's used should also be documented when data was created for a paper or report. And links to online information, papers and figures should be easy to include in the documentation.

These are the reasons why Sphinx is chosen for documenting our projects.

Sphinx introduction

Sphinx is a document markup language based on the reStructuredText (short: RST or reST) subsystem of the Docutils project, a project to process plaintext documents to a variety of formats like HMTL, XML and TeX.

It was originally build for documenting Python modules, but has been used in other projects and is now the new standard for documenting the Linux kernel source.

Installation and configuration

Installing the Sphinx package under Ubuntu is done by entering:

```
~$ sudo apt install python-sphinx
```

To use sphinx, you need to run the sphinx quickstart installer in the project directory:

```
~$ cd example
~/example$ sphinx-quickstart
Welcome to the Sphinx 1.3.6 quickstart utility.

Please enter values for the following settings (just press Enter to
accept a default value, if one is given in brackets).

Enter the root path for documentation.
> Root path for the documentation [.]: sphinx
```

The above root path puts the sphinx documentation files in a separate '`sphinx`' directory, which we recommend for clarity. Most of the other default values can be accepted, but there are a few questions which need to be entered manually, such as the project name, version and author:

```
The project name will occur in several places in the built
documentation.
> Project name: example
> Author name(s): Frank Everdij

Sphinx has the notion of a "version" and a "release" for the
software. Each version can have multiple releases. For example, for
Python the version is something like 2.5 or 3.0, while the release is
something like 2.5.1 or 3.0a1.  If you don't need this dual structure,
just set both to the same value.
> Project version: 1.0
> Project release [1.0]:
```

For using formulas in your documentation, answer one of the questions with `[y]`:

```
> pngmath: include math, rendered as PNG images (y/n) [n]:
> mathjax: include math, rendered in the browser by MathJax (y/n) [n]:
```

`pngmath` will render the formula as a `png` image, but this requires Sphinx version 1.4 and a local LaTeX implementation. Currently, Ubuntu 16.04 does not supply this particular version of Sphinx by default.

`mathjax` allows placing `TeX` formula code into RST documents and are rendered in HTML pages, but this requires an internet connection.

Once the quickstart is complete, the Sphinx directory should look like this:

```
~/sphinx$ ls -F
_build/  conf.py  index.rst  Makefile  _static/  _templates/
```

The `index.rst` is the main file which you need to edit in order to create your documentation. Additional content you require for creating the document such as images can be put in the `_static` directory.

The `_build` directory is where Sphinx stores the output after running `make html` or `make latex` . In the case of HTML, Sphinx will create the file `_build/html/index.html` which can be viewed locally in the browser.

The `_templates` directory is only used for changing the layout of HTML or LaTeX(PDF) output: Sphinx has default templates which should be sufficient for our purposes.

`conf.py` is the configuration script in Python for the Sphinx directory. Changes in default values and extensions to Sphinx should be edited in this file.

Syntax

The reStructuredText markup syntax is based on ASCII text with special meaning to some combination of characters, separated by either non-word characters such as spaces or tabs or empty lines.

Highlighting

Text highlighting or accentuation is done by enclosing characters:

Single asterix for italic : `*italic*` = *italic*

Double asterix for bold : `**bold**` = **bold**

Double backquotes for non-formatted or verbatim text : ``` ``verbatim`` ``` = `verbatim`

Single backquotes followed by an underscore represents a hyperlink and is very useful for the HTML output of the sphinx document in linking internal objects and webpages:

`` `Postprocessing`_ `` = `<a href="index.html#postprocessing">`

This indicates that the Sphinx HTML build will create a hyperlink to the chapter or section with the caption “Postprocessing”. Sphinx will automatically search and find the correct caption and make a link referring to that caption.

Chapters and sections

Captioned chapters, sections and paragraphs can be marked with symbols underlining the captioned word or words. For example, a Chapter called “Documentation” followed by two sections named “Background” and “Code” can be represented as follows:

```
Documentation
=====================================

Background
++++++++++++++++++++++++

Code
++++++++++++++++++++++++
```

The underlying symbols should at least be as many as the captioned word or words. There is no fixed convention what symbol should be used for which level. Sphinx will automatically figure out the level nesting and adjust the layout of these captions to reflect the level structure.

Within chapters or sections, paragraphs need to be separated by an empty line. If the empty line is absent, Sphinx will assume that the text is a continuation of the previous line and will resume where the previous line ended.

Lists

To format a list, use `*` for bulleted or `1.` , `2.` , etc for numbered lists:

```
* first line
* second line
```

gives:

- first line
- second line

Text blocks

C Code blocks can be formatted using the directive ‘ `.. code-block:: C` ‘ followed by a text block paragraph. The paragraph block must be indented and must have an empty line before and after the block. An example:

```
This is a C-code block:

.. code-block:: C

  printf(“hello world!\n”);
  return;
```

gives:

This is a C-code block:

```
printf(“hello world!\n”);
return;
```

Note the empty line between the first text line and the directive. An easier way to achieve the same result is:

```
This is a C-code block::

  printf(“hello world!\n”);
  return;
```

… which produces the same output.

Math formulas

If a Math extension is included and configured, math formulas with LaTeX equation syntax can be included:

```
:math:`\underline{x}=[x_{1}, ...,  x_{n}]^{T}`
```

This will be displayed as:

$$\underline{x} = [x_1, \ldots, x_n]^T$$

Images

To include an image, use:

```
.. image:: _static/image.png
   :align: center
```

as directive syntax. If you want to caption the image and add text, use `figure` :

```
.. figure:: _static/figure.png
   :align: center

   figure text in here
```

Note the indentation for the figure text caption.

Movies

Movies can also be included, either as a movie file with the ‘ `.. raw:: html` ‘ directive:

```
.. raw:: html

    <video controls src="_static/SampleVideo_360x240_1mb.mp4"></video>
```

Or as a link to a Youtube movie:

```
.. youtube:: JtqmDZrpfCA
```

The Youtube directive needs `python-sphinxcontrib.youtube` to be installed and configured in `conf.py`.

Unit test

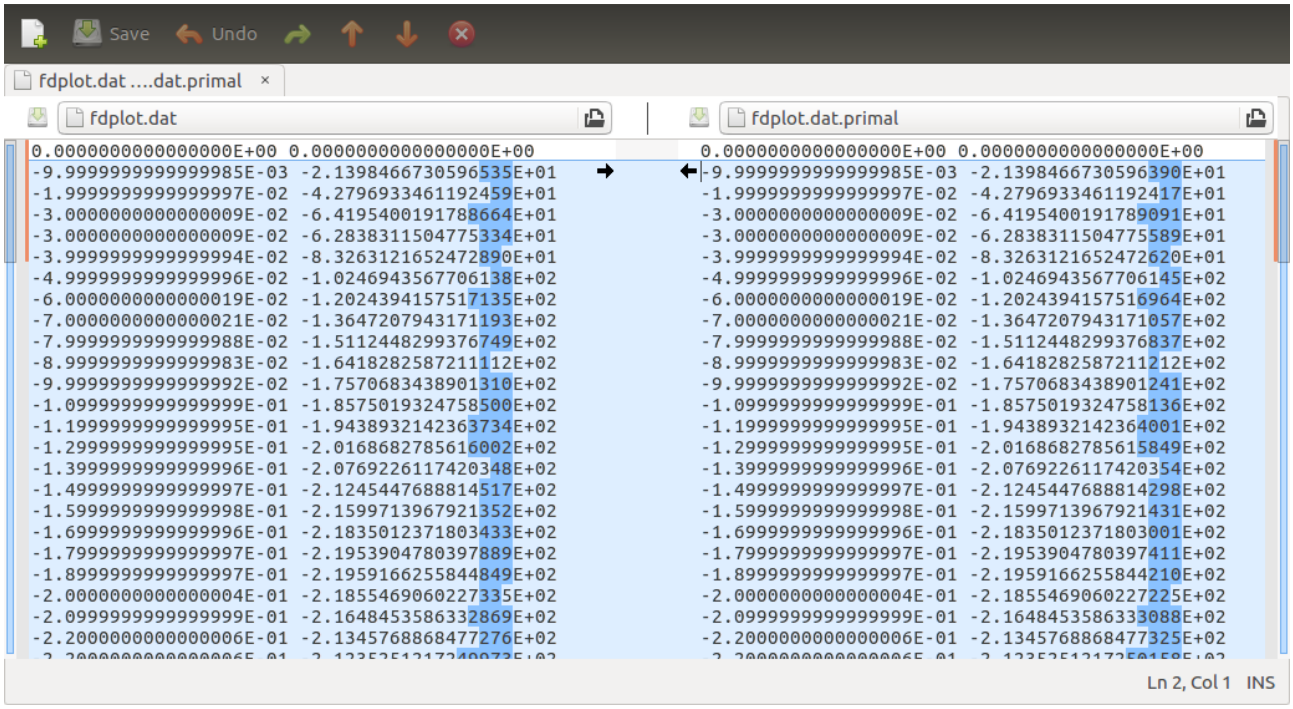
A Unit test is a test of a single program unit or function, usually performed by execution of a program and comparing an internal or external result with a reference value. This is carried out to ensure that (parts of) a program continue to function properly after a change in program code, compiler version, input data or change in external libraries.

We are going to use unit-testing not only for finding errors in program and individual units but also to provide a reference for projects which are to be archived after completion. This way we can ensure that given the code, documentation and unit test reference files we can build, run and verify the program being archived in the project.

Traditionally, unit tests were mostly performed by comparing results with previous verified results, basically treating the entire program as a unit. Recently there are libraries which integrate unit testing in a program framework, such as PyUnit for Python or CppTest or UnitTest++ for C++.

Since there is no unit test framework integrated in JemJive software, our use of unit-testing programs using JemJive libraries will be limited to comparison of output data (and optionally, verification of input data).

To compare output data with a reference data a simple difference program such as `diff` could suffice. However, there is the problem of comparing floating point numerical results. Below is a screenshot of a graphical difference program comparing two similar, but not identical data files:



As you can see both second columns start to differ in the 14th decimal. There can be many reasons for numerical differences as seen above:

- Using different input parameters for a certain problem. By altering timestep sizes or tolerances certain algorithms produce slightly different results.
- Modifying an algorithm by reordering terms. This modified accumulation of terms or quantities changes the numerical precision.
- Performing the calculations in parallel instead of serial code can also lead to the above situation, because of changes in numerical algorithms.
- Performing the calculation on a different computer: Some processors have an alternate, different flow of instructions and data or use different instructions for performing certain tasks.
- Using different compiler options: Some optimalisation options such as `-O3` will alter precision of intermediate and final results.
- Linking with a different library: some libraries are optimized for speed, which can affect numerical precision of the outcome of functions used in the program.

These sources of numerical differences can often be mitigated or avoided, but not ruled out entirely. Especially if a project needs to be archived for later use on different hardware and software, or a reference implementation is tested on many types of different machine configurations and compilers, these differences will occur.

To quickly test if a program has obtained the correct result, a certain error margin or ‘bandwidth’ should be used to determine if the difference of a numerical quantity with its reference value falls within a certain range. This can be done in an absolute or relative fashion:

Absolute: $|a-b| \leq \textit{absolute error}$

Relative: $\frac{|a-b|}{a} \leq \textit{relative error}$

To facilitate this comparison of floating point numbers in files, the program `numdiff` is suitable for comparing these data files. The program has options for using either relative or absolute errors, or for matching files with different layout or format of the numerical data.

A Python script is written around `numdiff` to make it easier for users to perform comparisons on data files in different directories. It is called `autounit.py` and is distributed with the project example repository as part of the next chapter: Project Directory.

autounit.py

In order for the script to work you need to install `python3` and `numdiff`:

```
~$ sudo apt install python3 numdiff
```

It has a help page when given the option `-h` :

```
~$ ./autounit.py -h
usage: autounit.py [-h] [-b] [-e] [-d] [-u] [-o] [-a] [error]

Performs automatic recursive operations on a JemJive project tree

positional arguments:
  error             value for relative or absolute error (default: 1e-06)

optional arguments:
  -h, --help        show this help message and exit
  -b, --build       perform program builds (default: False)
  -e, --execute     execute programs with -opt extension (default: False)
  -d, --docbuild    perform sphinx document builds (default: False)
  -u, --unittest    perform unittest on output files (default: False)
  -o, --onlyinunit  only build/execute when unittest files are present
                    (default: False)
  -a, --abserror    use absolute error instead of relative in unittest
                    (default: False)
```

For unit-testing we are using the `-u` option and the other relevant options are `error` for changing the default relative error number and `-a` to indicate the use of an absolute error criteria.

Use

The script will recursively scan the current directory for files with the extension `.unittest` or directories with name `unittest` :

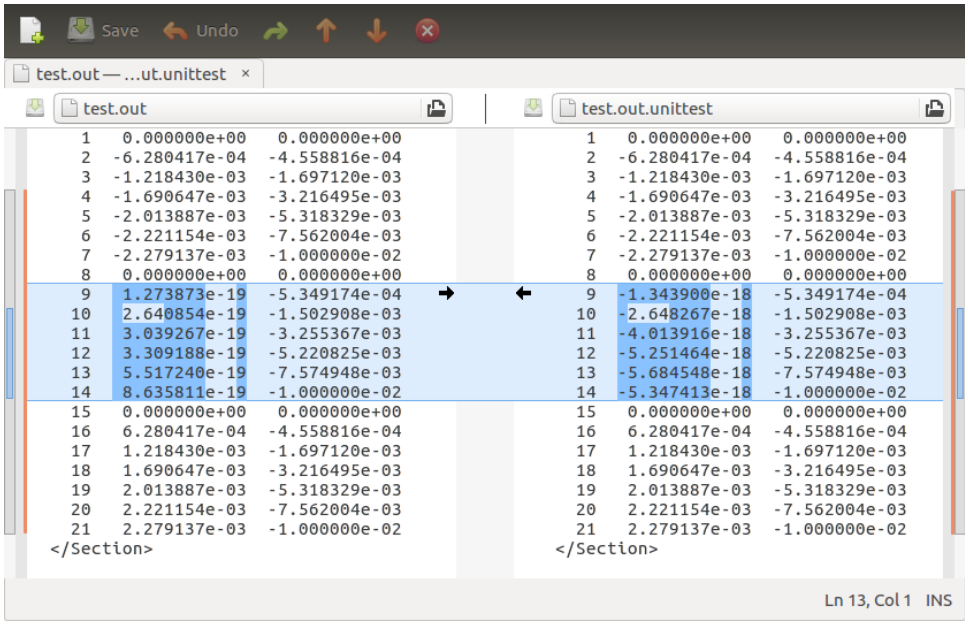
If it finds a file with `.unittest` extension, it tries to find a matching filename without the `.unittest` extension and if found, automatically performs a `numdiff` on these files:

```
~$ tree example
example
├── test.log
├── test.out
└── test.out.unittest

~$ ./autounit.py -u
    unittest: In ./example
 +++ Files test.out and test.out.unittest match.
```

If it finds a directory with name `unittest` it tries to match all files within that directory with filenames in the current directory and performs a `numdiff` on each of those files:

```
~$ tree example
example
├── test.log
├── test.out
└── unittest
    └── test.out

~$ ./autounit.py -u
    unittest: In ./example
 +++ Files test.out and unittest/test.out match.
```

For some files performing a `numdiff` with absolute error makes more sense. Consider this example:



With relative error as the default, the script will fail on unit-testing this file pair.

```
~/test$ ../autounit.py -u
    unittest: In .
 --- Files test.out and test.out.unittest differ by more than 1e-06
relative error.
```

This is due to the deviating values being situated around zero and having opposite sign, which is not suitable for using relative error criteria. From inspection we can see that those values are very small compared to the rest, about a factor 10^{-14} . Switching to absolute error criteria and reducing the error value to 10^{-16} gives:

```
~/test$ ../autounit.py -ua 1e-16
    unittest: In .
 +++ Files test.out and test.out.unittest match.
```

Test tips

Documenting which criteria should be used on which pair of files of the unit-testable results is important: It makes sure the tests are repeatable and reliable.

Make sure to unit-test the files you use for producing plots for a paper or article. This does not only increase the chance that your plots can be recreated at a later time, but also provides a backup of the data files used in making the plots.

Be aware that not every output file is suitable for unit testing. Files with date and time stamps or displaying performance statistics of certain functions produce different output at different times and platforms. Log files are a good example of those files.

Other script uses

In combination with a standard directory structure as proposed in the next chapter, the script can also perform builds and execution of JemJive code projects and creating the HTML output of the Sphinx documentation:

- The `-b` (build) option recursively searches the current directory for `src/Makefile` and attempts a `make opt` in `src` with the purpose of creating a program binary with the extension `-opt`, indicative of an optimized JemJive binary.
- The `-d` (document) option recursively searches the current directory for `sphinx/Makefile` and attempts a `make html` in `sphinx` for creating the HTML output of the Sphinx documentation.
- The `-e` (execute) option recursively searches the current directory for JemJive input files with extension `.pro` . If found, it searches for an executable program binary with the `-opt` extension and runs the JemJive computation with that binary in the same directory as the input file.

If only a build and execute of directories containing unittest files is required, the `-o` option can be added to achieve this. It will save time by skipping directories without unittest files.

Project directory

A project directory should be aptly named and uniquely identifiable. The use of lowercase with underscores as separators is preferred: Using spaces in directory names is strongly discouraged.

In the project directory, all files relevant to running a program, creation and analysis of project data and writing a report should be present. This means we should choose a directory for program code, a directory for project documentation and one or several directories for computation by running the program in each directory with different parameters.

A minimal directory structure with files would be, reflecting the above constraints:

```
<project_name>/
├── <name_of_computation>/
├── <report_name>/
├── sphinx/
│   ├── Makefile
│   ├── index.rst
│   └── conf.py
└── src/
    └── Makefile
```

Directories containing computation should contain all necessary files to run the computation. This way one can make more than one computation directory for different tasks: parameter studies, larger meshes or different models.

The Sphinx directory `sphinx` contains a `Makefile` for the various output format options which Sphinx support, the `index.rst` file which contains the main documentation for the project, and the `conf.py` Python script which holds the default values and extensions needed for creating the documents.

The Source directory `src` should contain a `Makefile` which enables the build of a single executable with the command `make`. For JemJive code this is the default and for most other programming languages as well. If other build-tools or scripts are necessary, do mention this in the Sphinx documentation.

Example project

The repository `example-project` contains a project directory with JemJive code, documentation and computation directories to create data and figures for the paper "*Cohesive modeling of transverse cracking in laminates under in-plane loading with a single layer of elements per ply*" by F.P. van der Meer and C.G. Dávila.

Users can clone this repository by entering the command:

```
$ hg clone ssh://sissy.tudelft.net/example-project example-project
```

It consists of a directory `fpm13ijss` which is the project of Dávila and Van der Meer and a Python script called `autounit.py` which will be explained later in unit-testing.

A tree view of all directories of the `fpm13ijss` project directory:

```
fpm13ijss/
├── CROSSMAN
│   ├── multiple
│   │   ├── g170
│   │   │   ├── t0.25 ── t0.3 ── t0.4 ── t0.5 ── t0.6 ── t0.7 ── t0.8
│   │   │   └── t1.0 ── t1.2 ── t1.4 ── t2.0
│   │   └── g220
│   │       ├── t0.25 ── t0.3 ── t0.4 ── t0.5 ── t0.6 ── t0.7 ── t0.8
│   │       └── t1.0 ── t1.2 ── t1.4 ── t2.0
│   └── single
│       └── g170
│           ├── t0.25 ── t0.3 ── t0.4 ── t0.5 ── t0.6 ── t0.7 ── t0.8
│           └── t1.0 ── t1.2 ── t1.4 ── t2.0
├── matlab
├── NAIRN_2
│   ├── multiple
│   │   ├── 0 ── 1 ── 2 ── 3 ── 4 ── 5
│   └── single
│       └── 0 ── 1 ── 2 ── 3 ── 4 ── 5
├── sphinx
│   ├── _build
│   └── _static
├── src
│   └── OBJ_OPT
└── testProblem
```

The `CROSSMAN`, `NAIRN_2` and `testProblem` directories are for computation purposes and unit-testing the program. The `matlab` directory contains scripts for creating figures used in the paper. `sphinx` and `src` are the documentation and JemJive code directory, respectively.

Example documentation

Looking at the example sphinx document in `fpm13ijss` there are several chapters dealing with the code, computation and post-processing:

Background

If the project resulted in a paper or report, you need to include the DOI link for that paper in here, if it exists. You can also include a PDF of a manuscript and link it, if it is not accepted yet or not available on the web.

A short description of what the project is about is very insightful for readers not familiar with the specific topic, but keep it short. The linked or attached paper or report should contain al the information one needs of a certain project.

Code

Describe the code by giving a list of the code files (headers and source code files) and give a one sentence description of what function or subroutine each code file contains and what it does.

Also mention the compiler version and operating system used to create the code and the computer used to run the code.

Directory Structure

Explain what each computation directory represent: often there are sequenced subdirectories for parameter studies. Explain which parameter is varied in the directory sequence.

Input/output

For each computation, list the input file (or files) necessary to complete the computation and list the output it has produced. Auxilliary files for creating or modifying data or meshes should also be mentioned.

Postprocessing

Explain the postprocessing steps to create each figure in the paper. Make the steps automated and relative to the computation directory, since a cloned project has a completely different path than the original. (Hint: test this!)

For some Matlab scripts which create figures it can be difficult to call other scripts in the project matlab directory, since Matlab's `addpath()` syntax relies on absolute paths. We successfully used:

```
addpath(fullfile(pwd,'..','..','..','matlab'));
```

at the beginning of a `.m` file as a workaround of specifying a relative path to the matlab directory three levels up.

Unit test

Document the unit tests which provide a meaningful test of the correctness of each computation: Do not unit test data which changes every run, like dates and execution times in a log. Better is to test an end-displacement-result or force-displacement plot, since that is what is often used in papers.

Example input file

If there are no comment section in the input file, it is very useful to explain the parameters and blocks contained within. This way, users can understand its structure and make modifications.

Appendix A: Secure Shell Access

To remotely access the mercurial-server on our cluster `sissy.tudelft.net` we will use agent forwarding to provide key authorization and tunneling in the case of doing work outside the university.

Enable agent forwarding

Every login session you need to manually start a key agent client, called `ssh-agent` which takes care of the private key file authorization.

First, start the key agent:

```
$ ssh-agent
```

Add the key:

```
$ ssh-add
```

You will be asked to type in your passphrase.

If the key file has no default name or is not in the default location, append `ssh-add` with the full file name:

```
$ ssh-add ~/.ssh/feverdij.rsa
```

The agent will then forward your private key to our cluster, so you don't have to worry about authorization during your session.

NOTE: The following documented steps for accessing the cluster via a bastion server assumes that you already have set up agent forwarding.

NOTE: In the following instructions please substitute the `<netid>` and `<loginname>` fields with the correct entries for your account.

Basic access

If agent forwarding has been setup and your location is inside the University, you can use Mercurial commands for remote services like:

```
$ hg clone ssh://hg@sissy.tudelft.net/feverdij/example example
```

If the default path for Mercurial is set in `.hg/hgrc` , `hg pull` and `hg push` do not need path arguments anymore.

If no `ssh-agent` is running, you are requested to type in your passphrase for your private keyfile.

Access via bastion server

The bastion server is a machine designed to act as a relay between the outside world and the University network. Its name is `linux-bastion.tudelft.nl`

Because Mercurial uses the `ssh` command protocol for remote repository operations like `hg clone` and no explicit shell connection is made, the only way to interact with the mercurial server on `sissy.tudelft.net` from outside the university is by means of a ssh-tunnel.

Tunneling

First make sure that agent forwarding is setup via the instructions above.

Then open a connection to the bastion, without opening a shell, but creating a tunnel to the cluster:

```
$ ssh -NA <netid>@linux-bastion.tudelft.nl
      -L22022:sissy.tudelft.net:22 &
```

This `ssh` command basically connects port `22022` on your system with the ssh-port `22` on the cluster. Because of the ampersand '`&`' the `ssh` command will run in the background so the shell is free for input again. Do not close the shell however, since the `ssh` command still depends on it and will abort if you do that.

Now open a second terminal (or re-use the freed shell terminal) and use your remote repository command from Mercurial, just as in the **Basic access** example:

```
$ hg clone ssh://hg@localhost:22022/feverdij/example example
```

This `hg clone command` via `ssh` will connect to the local port `22022` on your local machine. But because of the tunnel set up by the former `ssh` command, the packages will travel through the tunnel and connect with the cluster on the other side.

The '`config`' file

All of this is a lot of typing, and nobody can memorize these string of commands, but you can automate it a lot by using a file called `config` in the `~/.ssh` directory. Your `config` file would look something like:

```
# begin ~/.ssh/config
Host bastion
HostName linux-bastion.tudelft.nl
    User <netid>
    LocalForward 22022 sissy.tudelft.net:22
    Compression yes
    ForwardAgent yes
    ForwardX11 yes

Host sissy
HostName localhost
    User <loginname>
    Port 22022
     Compression yes
     ForwardAgent yes
    ForwardX11 yes
# end
```

Appendix B: Mercurial commands

This is a short overview of the most common Mercurial commands. Many commands support additional options to the command line. See **`man hg`** for a full list of commands and supported options.

Additionally, **`hg help <command>`** shows you the Python documentation of `hg <command>`.

Creation

`hg init <directory>`

> Create a Mercurial repository `.hg` in `<directory>`. If no directory is given, it creates the repository in the current directory.

Adding/Removing files

`hg add [<file>]`

> Adds `<file>` to be tracked by the repository. If no file is given, **`hg add`** will add all files that do not match glob or regexp patterns in `.hgignore`

`hg forget <file>`

> Untracks `<file>` in the repository, but do not remove `<file>` from the directory.

`hg remove <file>`

> Untracks `<file>` in the repository, and remove `<file>` from the directory.

Creating and undoing a revision

`hg commit [-m “text”]`

> Creates a new revision in the repository incorporating all changes in files. The option **`-m “<single line of text>”`** adds <single line of text> as a comment to the log entry of the revision. Without this option **`hg commit`** will open an editor to enter a comment. Not specifying a comment will abort the commit.

`hg rollback`

> Rolls back one repository transaction event, such as a successful commit.

Repository utilities on files

`hg diff [--rev # [--rev #]]`

Shows the edited changes between the files and the latest revision in `diff` style format. **`--rev`** shows the difference between the files and an earlier revision. **`--rev A --rev B`** shows the difference between the two revisions **`A`** and **`B`** .

`hg update [--rev #] [--clean]`

Updates all files in the repository to the current revision. Use **`--rev`** to update to an older revision. **`--clean`** disregards changes to files.

`hg revert <file> [--rev #] [--all]`

Reverts changes to `<file>` to the latest revision and creates a backup copy `<file>.orig` . Use **`--rev`** to revert to an older revision of the file. **`--all`** reverts all changed files.

Information

`hg status`

Shows the status of all the files which have been changed, added or removed or are created or are missing since the last revision.

`hg log [-l #] [--rev #]`

Shows the list of all revisions in a repository. To limit the output to a certain number of recent revisions, use **`-l`**. **`--rev`** shows the log for a specific revision.

`hg tip`

Alias for **`hg log --rev “tip”`**

Copying and synchronizing repositories

`hg clone <srcdir> <destdir>`

Clones the repository from `<srcdir>` to `<destdir>` and updates the `<destdir>` directory with files from the current active revision. If `<destdir>` is not present, it will be created.

`hg pull <srcdir>`

Copies changes from the repository in `<srcdir>` to the current repository, provided that they are related. If **`hg pull`** creates a new revision head, it attempts to merge the heads. **`hg pull`** does not modify files.

`hg incoming <srcdir>`

Same as **`hg pull`** but only lists the changes from the remote repository.

`hg push <srcdir>`

Copies changes from the local repository to the remote repository in `<srcdir>`. Aborts if the changes create a new revision head.

`hg outgoing <srcdir>`

Same as **`hg push`** but only lists what changes will be sent to `<srcdir>`.

`hg merge`

Opens the graphical merge utility to merge differences between files from local heads, if Mercurial cannot resolve the differences automatically. When the merge utility saves the results and finishes, the **`hg merge`** command is completed. Note that no new revision is created: Users must perform a **`hg commit`** after a merge.

References

“Version control concepts and best practices” by Michael Ernst:
https://homes.cs.washington.edu/~mernst/advice/version-control.html

“What is Version Control: Centralized vs. DVCS” by Giancarlo Lionetti:
http://blogs.atlassian.com/2012/02/version-control-centralized-dvcs/

“Introducing Mercurial to a small development team” by Norbert Kéri:
https://ihaveabackup.net/2011/12/04/introducing-mercurial-to-a-small-development-team

“Mercurial HowTo” by Angelo Simone:
ssh://hg@mech029.tudelft.net/~hg/repositories/Mercurial_howTo

Sphinx: Python Documentation Generator:
http://www.sphinx-doc.org/

reStructuredText: Markup Syntax and Parser Component of Docutils:
http://docutils.sourceforge.net/rst.html

Linux Kernel Documentation:
https://www.kernel.org/doc/html/latest/kernel-documentation.html#introduction

“Restructured Text (reST) and Sphinx CheatSheet” by Thomas Cokelaer:
http://openalea.gforge.inria.fr/doc/openalea/doc/_build/html/source/sphinx/rest_syntax.html

Tutorials

Official Mercurial Tutorial:
https://www.mercurial-scm.org/wiki/Tutorial

“A Mercurial tutorial” by Joel Spolsky:
http://hginit.com/

“Mercurial: The Definitive Guide” by Bryan O'Sullivan (old):
http://hgbook.red-bean.com/read/

Links

Python 2.7.12 Release:
https://www.python.org/downloads/release/python-2712/

TortoiseHG Download:
http://tortoisehg.bitbucket.org/download/index.html

Installing Sphinx:
http://www.sphinx-doc.org/en/1.4.8/install.html

PuTTY download page:
http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html

ToDo

- Windows support