Crash Course in C++

R F L Evans

www-users.york.ac.uk/~rfle500/

Course overview

- Lecture 1 - Introduction to C++
- Lecture 2 - Functions and Data
- Lecture 3 - Namespaces and Files
- Lecture 4 - Code Organization and Git
- Lecture 5 - Object Oriented Programming

Lecture 4
Code organisation and Git

- Multi-file projects
- Makefiles
- Git version control

Code spaghetti

- Imagine having a large project with 100,000 lines of code in a single file
- Each change to the source code requires recompiling everything
- Say you have a typo on line 45908 - you have to scroll through all those lines to get to it
- Then there is a related function on line 75652 so you have to scroll there to check the function arguments
- There is obviously a better way...

Multi-file Projects

- Projects with more than a handful of functions are often split between multiple files
- Typically one large or a few related functions per file
- Changing one file only requires that file to be recompiled
- Functions are easier to find and text editors often support several open files at once

Simple example

main.cpp

```cpp
#include <iostream>

int three(); // function declared here

int main(){

  int a = 4;

  std::cout << a+three() << std::endl;

  return 0;

}
```

three.cpp

```cpp
// function defined here
int three(){
  return 3;
}
```

Compiling multifile projects

```
g++ main.cpp three.cpp -o exe
```

- Include all files in the same compilation command
- Actually two processes together - compiling and linking
- Declaration of functions is needed for compilation
- Definition of function needed for linking

Large projects

- For programs with hundreds of functions, declaring all functions in every file tedious
- What if we need to change the argument list? Have to search for all occurrences of the function declaration
- Solution is to use a header file, which contains the function definition

Example header file

```
#ifndef FUNC_H_        // header guards prevent recursive inclusion
#define FUNC_H_
int three(); // function declaration
#endif
```

three.hpp

```
#include <iostream>
#include “three.hpp” // load header file
                     //(use “” for user defined headers)
int main(){
  int a = 4;
  std::cout << a+three() << std::endl;
  return 0;
}
```

main.cpp

```
// function defined here
int three(){
  return 3;
}
```

three.cpp

Large projects

- Modifications to function declaration only require changes in the header and source files, and wherever the function is called
- Compiler looks in same directory as source code by default
- Can also store header files in a different directory using -Idir compiler option

Variable sharing via header files

- Sometimes want to share a namespace between files
- Use the “extern” keyword to tell the compiler that this object is declared elsewhere
- Functions are always extern by default

Example namespace sharing

```
#ifndef NS_H_        // header guards prevent recursive inclusion
#define NS_H_
namespace ns{
  extern int a; // namespace variable
}
```

ns.hpp

```
#include <iostream>
#include “ns.hpp” // load header file

int main(){
  std::cout << ns::a << std::endl;
  return 0;
}
```

main.cpp

```
// function defined here
namespace ns{
  int a=3;
}
```

ns.cpp

Makefiles

- Projects with large number of files are cumbersome to recompile by hand
- Ideally want to separate compiling and linking so only modified files are recompiled
- Unchanged files are just re-linked - a much faster process
- Example - CASTEP takes over an hour for a full re-compilation
- Makefiles automate this process

Basic makefile

```
CC=g++ # Compiler
LIBS=-lstdc++ # Libraries
CC_CFLAGS=-O3 -mtune=native -I./hdr # Compilation flags
CC_LDFLAGS= -lstdc++ -I./hdr # Link flags

OBJECTS= obj/main.o obj/three.o # Object (source) files

EXECUTABLE=exe # Executable name

all: $(OBJECTS) serial

serial: $(OBJECTS)
    $(CC) $(CC_LDFLAGS) $(LIBS) $(OBJECTS) -o $(EXECUTABLE)

$(OBJECTS): obj/%.o: src/%.cpp
    $(CC) -c -o $@ $(CC_CFLAGS) $<

clean:
    @rm -f obj/*.o
```

Using make

- Need to include separate directories src, hdr and obj for source, header and object files
- Compile your project

```
make
```

- Clean all existing object files

```
make clean
```

- Force recompilation

```
make -B
```

- Compile using multiple CPUs (e.g. 4)

```
make -j 4
```

Introduction to Git

Development Process

- Want to add a major new feature to your code
- Others still use the existing version at the same time
- How do you merge the changes after the development?
- If you find a bug, how do you make sure both version are up to date?
- Use version control to manage the development process

What is git?

- Software to keep track of changes to files and manage them
- Specifically git is *distributed*; complete repository is kept on your local machine
- Essential in any modern software development
- Allows multiple ‘branches’ of a code to coexist peacefully

Getting started

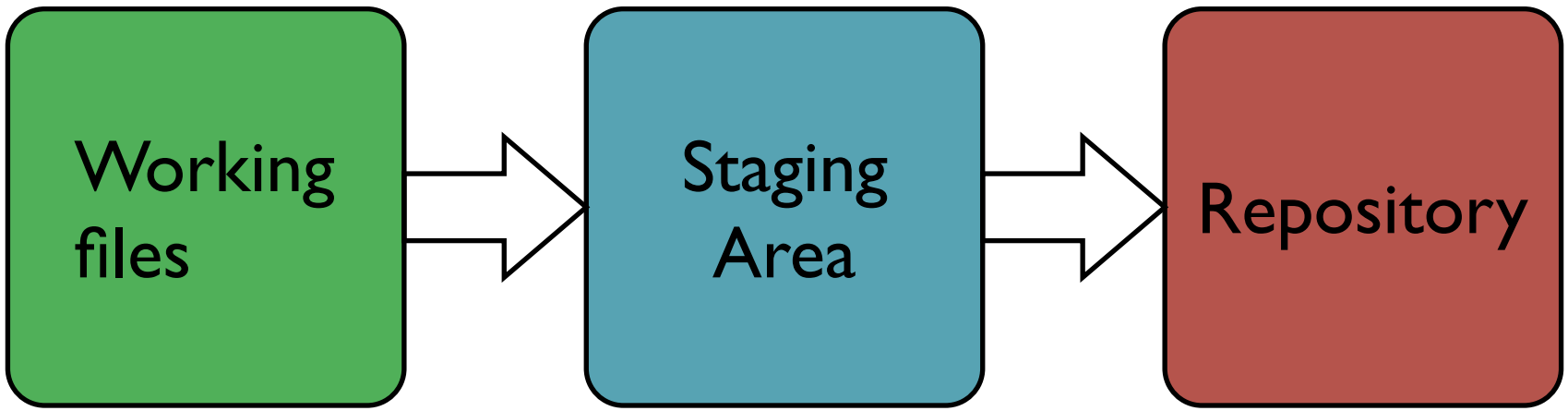
- http://schacon.github.com/git/gittutorial.html
- Identify yourself to git:

```
git config --global user.name "Joe Bloggs”
git config --global user.email ab@york.ac.uk
```

- Each project must be kept in its own folder

Creating a new project

```
mkdir myproject
cd myproject
git init              [Initialise git repository]
nano hello.cpp        [Create source file]
```



Adding files to the repository

- Stage file to staging area

```
git add hello.cpp
```

- Commit (save) file to repository

```
git commit -m “My first source file”
```

- Check status of repository

```
git status
```

- Check history of commits (q to finish)

```
git log
```

- Check working directory is clean before adding and committing files

Modifying files

- After changing file, add to staging area

```
git add hello.cpp
```

- Commit (save) file to repository

```
git commit -m “modified source file”
```

- If you have a modified file you can see the changes (before staging)

```
git diff
git diff hello.cpp
```

Git ignore

- Certain files should not be added to a source code repository - executables, scratch files, object files, result files
- Use .gitignore file to tell git which files to ignore

```
*.o
myexe
*.txt
# temporary files
*~
```

Git example

```
rm hello.cpp
```

- Oops...
- Normally an unrecoverable situation

```
git checkout hello.cpp
```

- Recovers last committed version of the file
- Always make lots of small commits, preferably that compile

Git branches

- Git handles multipe simultaneous versions of your code, called branches (of a tree)
- Can see which branches there are using:

```
git branch -a
```

- Typical output:

```
*master
```

- *indicates which branch you are working on, and *master* is the master branch where all commits are stored initially

Git branches

- Can add your own branches by creating them from the current branch

```
git checkout -b mybranch
```

- Makes a complete copy of your code and moves you to the new branch

```
branch -a
  master
*mybranch
```

- Development on this branch does not affect the master branch

Git branches

- Can switch between branches using:

```
git checkout master
git checkout mybranch
```

- Git will warn about uncommitted changes
- Always try to commit changes to files before switching branches
- Best way to develop new features
- Branching is easy and space efficient

Git merge

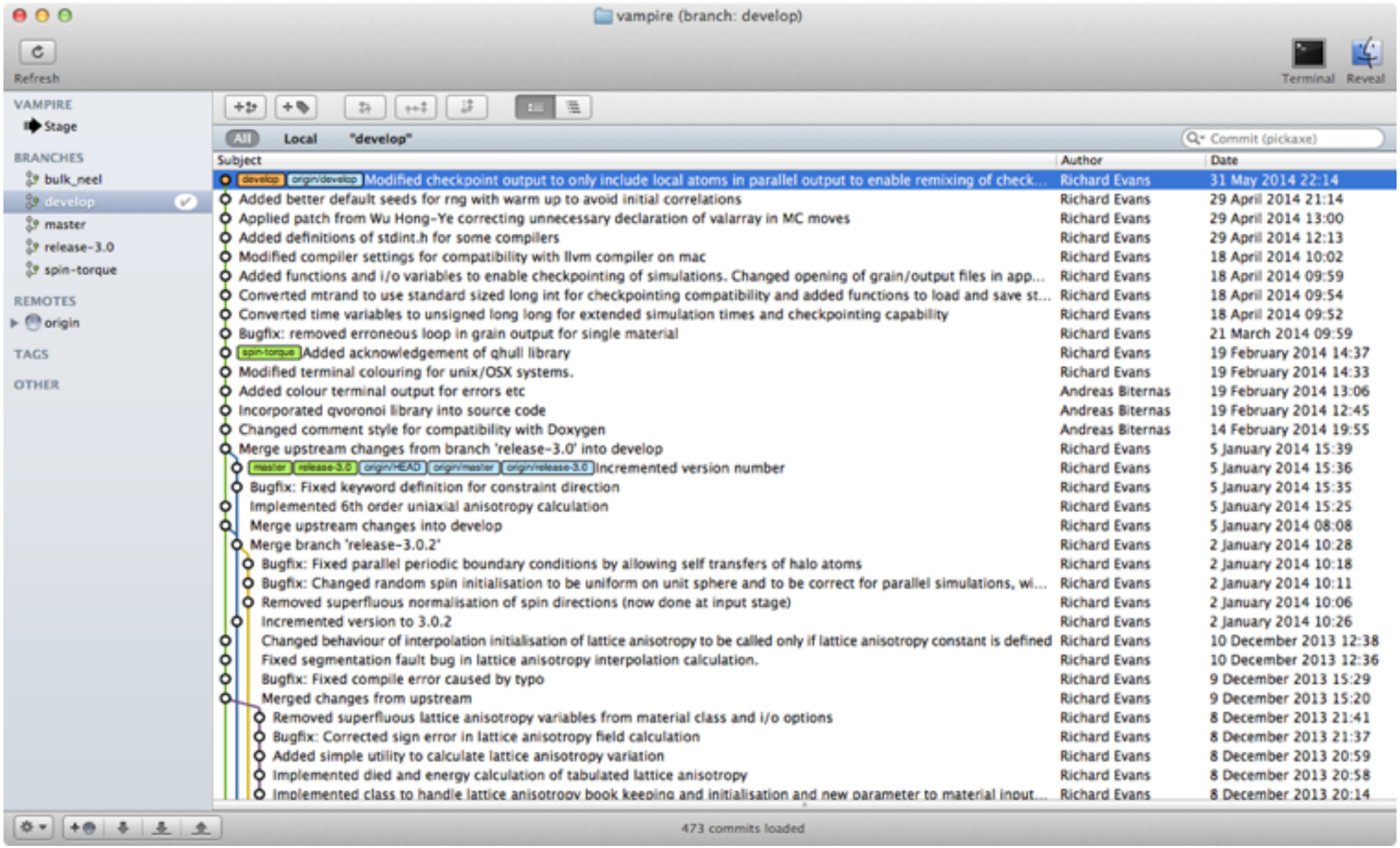
- Once your new feature is ready need to combine the old and new versions of the code
- Merging does this automagically (usually)
- Always ‘pull’ changes when merging

```
git checkout master
git merge mybranch
```

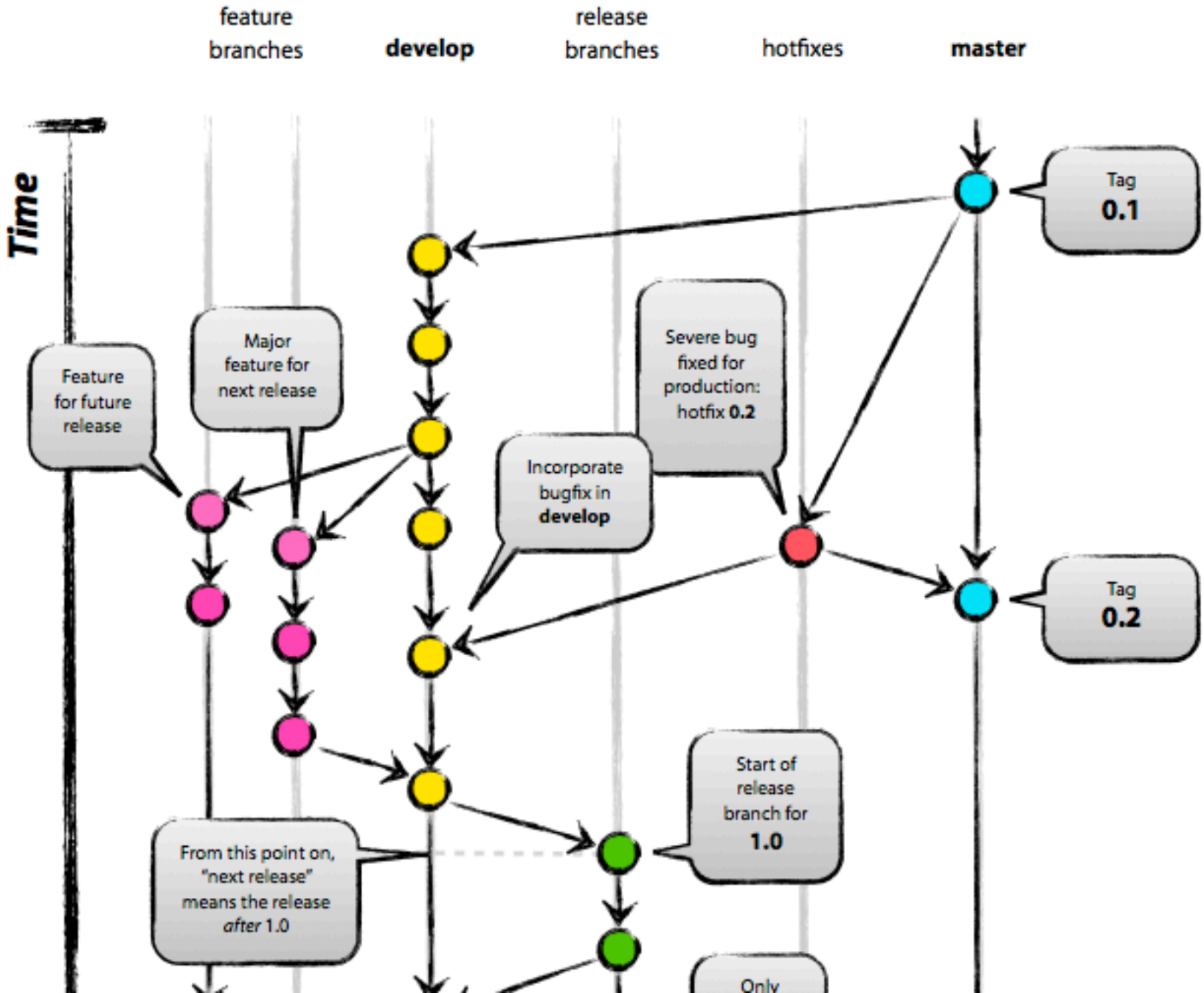
- Sometimes a file is modified in both branches - need to manually resolve conflicts - messy

Git workflow

- For small projects using branches is easy
- Larger projects generally require a more detailed structure - called a workflow
- Git is also well designed for distributed development - used in many open source projects
- Github.com is free for public repositories - great for storing code and managing projects
- Private repo's are available at cost


vampire (branch: develop)
Refresh
Terminal
Reveal
VAMPIRE
Stage
BRANCHES
bulk_neel
develop
master
release-3.0
spin-torque
REMOTES
origin
TAGS
OTHER
All
Local
"develop"
Commit (pickaxe)
Subject
Author
Date
develop
origin/develop
Modified checkpoint output to only include local atoms in parallel output to enable remixing of check...
Richard Evans
31 May 2014 22:14
Added better default seeds for rng with warm up to avoid initial correlations
Richard Evans
29 April 2014 21:14
Applied patch from Wu Hong-Ye correcting unnecessary declaration of valarray in MC moves
Richard Evans
29 April 2014 13:00
Added definitions of stdint.h for some compilers
Richard Evans
29 April 2014 12:13
Modified compiler settings for compatibility with llvm compiler on mac
Richard Evans
18 April 2014 10:02
Added functions and i/o variables to enable checkpointing of simulations. Changed opening of grain/output files in app...
Richard Evans
18 April 2014 09:59
Converted mtrand to use standard sized long int for checkpointing compatibility and added functions to load and save st...
Richard Evans
18 April 2014 09:54
Converted time variables to unsigned long long for extended simulation times and checkpointing capability
Richard Evans
18 April 2014 09:52
Bugfix: removed erroneous loop in grain output for single material
Richard Evans
21 March 2014 09:59
spin-torque
Added acknowledgement of qhull library
Richard Evans
19 February 2014 14:37
Modified terminal colouring for unix/OSX systems.
Richard Evans
19 February 2014 14:33
Added colour terminal output for errors etc
Andreas Biternas
19 February 2014 13:06
Incorporated qvoronoi library into source code
Andreas Biternas
19 February 2014 12:45
Changed comment style for compatibility with Doxygen
Andreas Biternas
14 February 2014 19:55
Merge upstream changes from branch 'release-3.0' into develop
Richard Evans
5 January 2014 15:39
master
release-3.0
origin/HEAD
origin/master
origin/release-3.0
Incremented version number
Richard Evans
5 January 2014 15:36
Bugfix: Fixed keyword definition for constraint direction
Richard Evans
5 January 2014 15:35
Implemented 6th order uniaxial anisotropy calculation
Richard Evans
5 January 2014 15:25
Merge upstream changes into develop
Richard Evans
5 January 2014 08:08
Merge branch 'release-3.0.2'
Richard Evans
2 January 2014 10:28
Bugfix: Fixed parallel periodic boundary conditions by allowing self transfers of halo atoms
Richard Evans
2 January 2014 10:18
Bugfix: Changed random spin initialisation to be uniform on unit sphere and to be correct for parallel simulations, wi...
Richard Evans
2 January 2014 10:11
Removed superfluous normalisation of spin directions (now done at input stage)
Richard Evans
2 January 2014 10:06
Incremented version to 3.0.2
Richard Evans
2 January 2014 10:26
Changed behaviour of interpolation initialisation of lattice anisotropy to be called only if lattice anisotropy constant is defined
Richard Evans
10 December 2013 12:38
Fixed segmentation fault bug in lattice anisotropy interpolation calculation.
Richard Evans
10 December 2013 12:36
Bugfix: Fixed compile error caused by typo
Richard Evans
9 December 2013 15:29
Merged changes from upstream
Richard Evans
9 December 2013 15:20
Removed superfluous lattice anisotropy variables from material class and i/o options
Richard Evans
8 December 2013 21:41
Bugfix: Corrected sign error in lattice anisotropy field calculation
Richard Evans
8 December 2013 21:37
Added simple utility to calculate lattice anisotropy variation
Richard Evans
8 December 2013 20:59
Implemented died and energy calculation of tabulated lattice anisotropy
Richard Evans
8 December 2013 20:58
Implemented class to handle lattice anisotropy book keeping and initialisation and new parameter to material input...
Richard Evans
8 December 2013 20:14
473 commits loaded

http://nvie.com/posts/a-successful-git-branching-model/



Practical time...