Online developer notes

The developer notes contain general information for developers that is useful no matter what part of the code you are editing. As such it does not provide arguments for or against specific design choices. For this type of information use the design decisions wiki instead. If you are interested in an introduction about how to use hpGEM, you may find the readme more useful.

About code conventions

The code conventions can be found on the website at http://www.hpgem.org/development/coding-style-of-the-kernel
They are intended to make the code more uniform. Uniform code is more easy to read. If you ignore the coding conventions, you may find that you suddenly have to resolve some mayor text conflicts or even tree conflicts, because somebody decided to clean up.

Committing conventions

If you commit things to the svn, please label your log messages, so it is easier to figure out what happened and what has to be ported to other branches of hpGEM. Please use one of the following labels

Label	Notes	Examples
New feature	• You added something to hpGEM that did not exist before • Your changes did not deprecate any existing routines • If this commit is part of a series of changes to the same feature, please say so in your commit • Once you feel your feature is finished please say so in your commit	• New feature: (finished) Added Runge-Kutta time integration to hpGEM • New feature: (in progress) Added support for parallel computations
Documentation	• You only changed comments and nothing else	• Documentation: added ASCII-art pictures of the reference geometries
Bug Fix	• You fixed a bug • If the bug was known please mention the JIRA job in the commit message and close the job • Includes performance bugs	• Bug-fix: assertion in Matrix::getRow did bounds checking on the number of columns • Bug-fix: (HPGEM-104) Wall boundaries were producing segfaults in APISimplified
Application	• You made changes to one or more of the application codes • Since you are probably maintaining the application, you know what you want to read in the commit log • But also see the notes on 'New feature' • Never commit actual data, even if it is just to move it from the cluster to your own computer	• Application: (back-up) 1D slope limiter probably works • Application: (in progress) Added directory structure for Navier-Stokes solvers • Application: (finished) Euler code works and passes the test cases
Interface change	• You made a set of changes that resulted in one or more routines now being deprecated • If you want to commit an interface change while the work is still in progress, please set up a dedicated branch in ^/branches/public/ • If you have to make changes to applications you did something wrong	• Interface change: Entrance classes for the API now start with HpgemAPI • Reintegrated: The mesh is now stored in a way that supports h-adaptation
Clean-up	• Your changes cannot be classified as any of the above	• Clean-up: Updated the classes in namespace Output for the new coding conventions

Before committing anything make sure your version is up to date and passes fullTest on your system. Structure your commits such that you need one label only. Multiple unrelated changes belong in different commits. Never rename (move) files.

About assertions

The intended use of assertions in general and logger.assert for hpGEM is to check blatant assumptions on the arguments provided to the functions. For example, if you are trying to request entry 12 of an array of size 4, this is clearly wrong and you should change your function.

Assertions should be used freely, but checking everything, all the time, is expensive. Therefore assertions are compiled out of the code completely in Release configuration. However, this also means that if you accidentally do something that has side effects in an assertion, the behaviour of your code may change between configurations. For example, NEVER DO 'logger.assert(x=4, "/// WRONG!!");'

Preparing data for the assertions is also expensive. If you want to assert something that depends on data that is not yet computed and not needed elsewhere in the function, do NOT compute it before the assertion. Rather write a dedicated function checkThisCondition that return bool and call it inside the assertion. (make sure to think of a better name for the function!)

About optimisation

Always assume compiler developers are better at optimising code than you are. They are hired specifically to optimise code. Moreover they don't have to worry about the readability of the code they produce and they can use tricks that are not available in non-compiled c++. Compiler developers have to make assumptions about the code they are going to process. These tend towards processing code that is easy to read and write. Because of this, if you break these assumptions, they will instead assume you know what you are doing and are after a specific effect. THIS MAY MEAN THAT WRITING CLEVER CODE BECAUSE YOU THINK IT IS FASTER PRODUCES WORST PERFORMANCE THAN A NAIVE IMPLEMENTATION!

That being said, there are things that can be done to make an application faster, you just have to be careful that they actually make the application faster.

Optimising code only makes sense if you are building in Release mode, before doing anything else, set CMAKE_BUILD_TYPE to Release and make sure hpGEM_ENFORCE_ASSERTIONS is OFF. All effort spend in other configurations is almost certainly wasted and probably even counterproductive. Make sure you have some timing information to compare before and after, to see if you made any improvements.

Before changing anything, you want to know where the bottlenecks are. To do this, use a profiler, such as gprof(linux) or Instruments(mac), on a representative application. With this information in hand either make sure the most expensive methods get called less often or try to improve the efficiency of this method. Note that saving information will most likely speed up your application, but comes at the cost of a larger memory footprint. This may mean large applications no longer run. This is NOT an ideal situation.

Between steps and after you are done, make sure to compare with your old timings. If things went worse, discard your changes and try again.

About debugging

Debugging code only makes sense if you are building in Debug mode, before doing anything else, set CMAKE_BUILD_TYPE to Debug. If this changes the behaviour of your code, first toggle hpGEM_DISABLE_ASSERTIONS to see if someone made a mistake in an assertion. If this did not alter the behaviour you have to hope a combination of CMAKE_BUILD_TYPE=RelWithDebInfo and hpGEM_ENFORCE_ASSERTIONS provides enough useful information

If you feel it is necessary to see additional debug output, you can set the advanced CMake option hpGEM_LOGLEVEL to DEBUG. There is quite a lot of additional debug info, so you probably want to pipe your output into grep or a similar text search tool.

If PETSc is reporting errors this usually means something went wrong in the interaction between hpGEM and PETSc. In this case it may be useful to add the PETSc option -on_error_abort. This will make your code crash on the first PETSc error.

If you are working on a mac, entering 'env DYLD_INSERT_LIBRARIES=/usr/lib/libgmalloc.dylib' in lldb before typing 'process launch' will enable lldb to trace memory corruption (for linux, valgrind can do this)

If something unexpected happens in the kernel of hpGEM this likely has one of the four following causes:

Case 1: Nonsense arguments were passed to a function and therefore it returned nonsense

This happens for example when some function tries to request entry 12 of an array of size 4. This situation should be debugged by adding assertions until the function successfully detects the erroneous arguments. In the example this means adding the line 'logger.assert(entry < 4, "This array only has 4 entries")'. Your code will now crash when 12 is passed to the array, making it more easy to find the places where stuff actually went wrong. Other developers may have the same issue, please commit the extra assertions.

Case 2: Some fields in a class contain nonsense and therefore it produces nonsense

This happens for example when someone constructs a face with a weird element. This means you are detecting one of the other two causes in an unfortunate place. Check the other functions that interact with the erroneous field for the actual source of the error.

Case 3: All arguments and the state of the class are valid, but it still breaks

This happens because of a programming error. Add a unit-test that detects the current problem and change things in the kernel until all unit tests pass. Once they do please commit the bugfix.

Case 4: Everything is fine, but there is an assertion or a unit test that fails

Unit-tests and assertions are great for detecting something is wrong, but unfortunately it may also be the unit-test or the assertion itself that is wrong. Please be extra careful to make sure it is actually the unit-test or the assertion that is wrong and when in doubt ask. When you commit the fix, clearly state in you commit message that you changed assertions or unit-tests because they were broken.

FAQ

Q: Your 'performance optimisations' made the test suite much slower

A: The test suite tries to test parts of hpGEM in isolation. This means it sometimes has to take some non-standard actions to create or manipulate data. It often also takes an action once for many different shapes/basisfunctions/etc. while a typical application takes this action many times for the same shapes/basisfunctions/etc.. This means performance considerations for the test suite are different from performance considerations for performance critical applications and improving speed for the latter may have adverse effect on the former.

Q: This bit of code seems like it is wasteful to me
A: Did you have a look at a profiler already? Did you read the section about performance optimisations already?