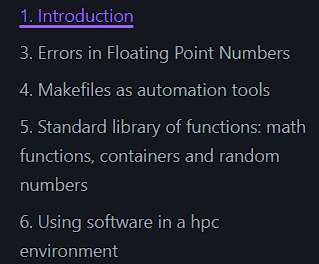
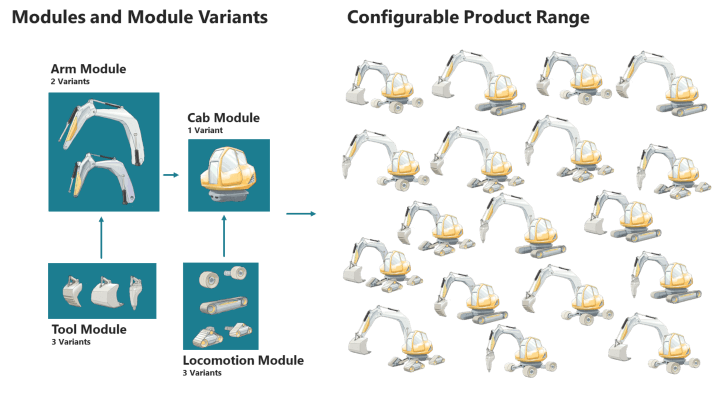
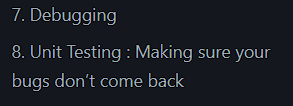
****

**Modularization:**

Is the activity of dividing a product or system into interchangeable modules. The target of modularization is to create a flexible system that enables the creation of different requested configurations, while also reducing the number of unique building blocks (module variants) needed to do so.



In the software context Modularization is the process of separating the functionality of a program into independent, interchangeable modules, such that each contains everything necessary to execute only one aspect of the desired functionality.

****

**Debugging**

Program debugging is the process of identifying and correcting programming errors.

Anybody who has written software for public use will probably have received at least one bad bug report. Reports that say nothing ("It doesn't work!"); reports that make no sense; reports that don't give enough information; reports that give wrong information. Reports of problems that turn out to be user error; reports of problems that turn out to be the fault of somebody else's program; reports of problems that turn out to be network failures.

In bug reports, try to make very clear what are actual facts ("I was at the computer and this happened") and what are speculations ("I think the problem might be this"). Leave out speculations if you want to, but don't leave out facts.

It's the same with programmers. Providing your own diagnosis might be helpful sometimes, but always state the symptoms. The diagnosis is an optional extra, and not an alternative to giving the symptoms. Equally, sending a modification to the code to fix the problem is a useful addition to a bug report but not an adequate substitute for one.

### **Summary:**

* The first aim of a bug report is to let the programmer see the failure with their own eyes. If you can't be with them to make it fail in front of them, give them detailed instructions so that they can make it fail for themselves.
* In case the first aim doesn't succeed, and the programmer can't see it failing themselves, the second aim of a bug report is to describe what went wrong. Describe everything in detail. State what you saw, and also state what you expected to see. Write down the error messages, especially if they have numbers in.
* When your computer does something unexpected, freeze. Do nothing until you're calm, and don't do anything that you think might be dangerous.
* By all means try to diagnose the fault yourself if you think you can, but if you do, you should still report the symptoms as well.
* Be ready to provide extra information if the programmer needs it. If they didn't need it, they wouldn't be asking for it. They aren't being deliberately awkward. Have version numbers at your fingertips, because they will probably be needed.
* Write clearly. Say what you mean, and make sure it can't be misinterpreted.
* Above all, be precise. Programmers like precision.

**List of Bugs:**

Many programs, particularly free ones, publish their list of known bugs. If you can find a list of known bugs, it's worth reading it to see if the bug you've just found is already known or not. If it's already known, it probably isn't worth reporting again, but if you think you have more information than the report in the bug list, you might want to contact the programmer anyway. They might be able to fix the bug more easily if you can give them information they didn't already have.

* **Debugging with GDB: https://news.ycombinator.com/item?id=30512302**
* **Debugging with PDB (Python debugging):** [**https://www.youtube.com/watch?v=a7qIcIaL4zs**](https://www.youtube.com/watch?v=a7qIcIaL4zs)
* **gdb breakpoint tricks: https://news.ycombinator.com/item?id=39170901**
* **Reverse engineering: hugsy/gef**

**Testing:**

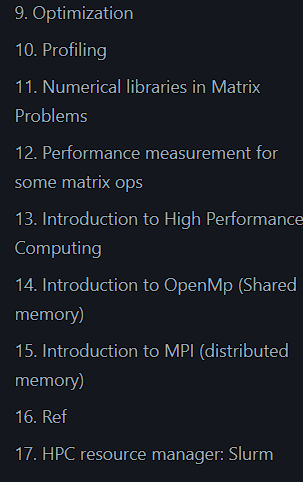
Unit testing allows to ensure that a given software behaves in the correct way, at least for the cases one is testing. Once a function is written (or even before in TTD (Test-Driven Development)) or a bug is fixed, it is necessary to write a test that ensures the function to work properly in limit cases or the bug to not reappear in the future. There are several levels associated with unit testing.

It is worth mentioning that catching an exception (try and catch blocks) also helps with handling runtime errors and reacting correspondingly. Also, using logging libraries like logger or loguru (python), or spdlog, https://github.com/gabime/spdlog, (c++) allows to print useful logging messages that also helps when you try to understand what is going on with your program.

* **Test-Driven Development:**

Is a process for when you write and run your tests. Following it makes it possible to have a very high test-coverage. Test-coverage refers to the percentage of your code that is tested automatically, so a higher number is better. TDD also reduces the likelihood of having bugs in your tests, which can otherwise be difficult to track down.

* **Unit Testing:**Unit testing allows to ensure that a given software behaves in the correct way, at least for the cases one is testing. Once a function is written (or even before in TTD) or a bug is fixed, it is necessary to write a test that ensures the function to work properly in limit cases or the bug to not reappear in the future. There are several levels associated with unit testing .
* **Integral Testing:**

****

**High Performance Computing:**

First, make your program correct, then measure, then optimize. Optimization can be achieved by just using compiler flags, by using already optimized libraries, and sometimes by being careful with our memory usage. Some low level techniques are better left to the compiler.

**Profiling:**

You need to detect functions which take most of the time. Optimizing a function that takes only 5% of the time will give you only marginal benefits. Finding the functions that take most of the time is called profiling , and there are several tools ready to help you.

* **Measuring the whole running time:**
* **Measuring elapsed time:**

The first approach is to just add timers to your code. This is a good practice and it is useful for a code to report the time spent on different parts. We will use the previous example and add watches at specific pointsm, using std::chrono,you can see more examples:

* https://www.techiedelight.com/measure-elapsed-time-program-chrono-library/
* https://en.cppreference.com/w/cpp/chrono/duration.
* **Profilers:**

There are many types of profilers from many different sources, commonly, a profiler is associated to a compiler, so that for example, GNU (the community around gcc compiler) has the profiler ‘gprof’, intel (corporation behind icc) has iprof, PGI has pgprof, etc. Valgrind is also a useful profiler through the cachegrind tool, which has been shown at the debugging section.

**Benchmarking:**

In this section we will learn how to perform some benchmarks on some matrix operations to get an idea of the operations performance. This will be done manually, but in the future you can use specialized tools like google benchmark, google/benchmark, or nanobench, https://nanobench.ankerl.com/ , etc.

**Speedup:**

**S(N) = T (s,1)/T(s,N)**

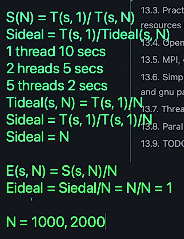
**T\_ideal =T(s,1)/N**

**S\_ideal = T (s,1)/T\_ideal (s,N) =N**

**Eficiencia paralelaSeedup**

**E(s,N) = S(s,N)/N**

**E\_ideal = S\_ideal/N = N/N =1**



**Good programming practices:**

**Obs:**

