FeenoX programming guide

2025-06-17

Contents

1 Why we program FeenoX

The main objective is to comply with the Software Requirements Specification. Within this goal, there are three levels of importance:

1. FeenoX is Free Software as defined by the Free Software Foundation. The four essential freedoms ought to be assured. Feenox is GPLv3+ so make sure you understand what "free-as-in-free-speech"

means and all the details that the license implies regarding using other libraries and code copyright.
2. FeenoX is Open Source as defined by The Open Source initiative. Stick to open standards, formats, practices, etc. Make sure you understand what the difference between "free software" and "open source" is.
3. Only after making sure every piece of code meets the two criteria above, the technical and/or efficiency aspects mentioned in the SRS are to be considered.

2 Compiling and debugging

See the compilation instructions.

3 How we program FeenoX

3.1 Operating systems

- The main target operating system is GNU/Linux, particularly `linux-x86_64`.
- The reference distribution is Debian stable. All the required dependencies should be available in any decently-modern Debian repositories.
- Support for other architectures (mainly the other two non-server end-user mainstream OSes which are non-free and we do not want to mention explicitly) is encouraged but not required.

3.2 Languages

> Rule of Simplicity: Design for simplicity; add complexity only where you must.

C++ is a great language when used to solve problems it solves naturally well without being forced to solve it (see LibreBlackjack). C++ is a terrible language whenever else. It is messy, overwhelming, complicated, not robust and hard to debug (especially when using templates and this nightmare called smart pointers). The cons are far more than the pros. There is no need to add complexity.

Fortran is a terrible language. Fortran90+ is a patched FORTRAN 77. The assumptions that the language had over the kind of computers it has to run on worked in the early days but are now widely outdated. That's it.

Go & Rust, I think still nobody knows for sure right now. But Rust's automatic memory handling is not a point strong enough for CAE/DAE/FEA software.

C is the best language for FeenoX. Its assumptions still hold, especially in

i. bare-metal servers,
ii. virtualized machines, and
iii. dockerized containers.

> In the late 1990s, Gerrit Blaauw and Fred Brooks observed in Computer Architecture: Concepts and Evolution [BlaauwBrooks] that the architectures in every generation of computers, from early mainframes through minicomputers through workstations through PCs, had tended to converge. The later a design was in its technology generation, the more closely it approximated what Blaauw & Brooks called the "classical architecture": binary representation, flat address space, a distinction between memory and working store (registers), general-purpose registers, address resolution to fixed-length bytes, two-address

instructions, big-endianness,[46] and data types a consistent set with sizes a multiple of either 4 or 6 bits (the 6-bit families are now extinct).

Thompson and Ritchie designed C to be a sort of structured assembler for an idealized processor and memory architecture that they expected could be efficiently modeled on most conventional computers. By happy accident, their model for the idealized processor was the PDP-11, a particularly mature and elegant minicomputer design that closely approximated Blaauw & Brooks's classical architecture. By good judgment, Thompson and Ritchie declined to wire into their language most of the few traits (such as little-endian byte order) where the PDP-11 didn't match it.[47]

The PDP-11 became an important model for the following generations of microprocessor architectures. The basic abstractions of C turned out to capture the classical architecture rather neatly. Thus, C started out as a good fit for microprocessors and, rather than becoming irrelevant as its assumptions fell out of date, actually became a better fit as hardware converged more closely on the classical architecture. One notable example of this convergence was when Intel's 386, with its large flat memory-address space, replaced the 286's awkward segmented-memory addressing after 1985; pure C was actually a better fit for the 386 than it had been for the 286.

It is not a coincidence that the experimental era in computer architectures ended in the mid-1980s at the same time that C (and its close descendant C++) were sweeping all before them as general-purpose programming languages. C, designed as a thin but flexible layer over the classical architecture, looks with two decades' additional perspective like almost the best possible design for the structured-assembler niche it was intended to fill. In addition to compactness, orthogonality, and detachment (from the machine architecture on which it was originally designed), it also has the important quality of transparency that we will discuss in Chapter 6. The few language designs since that are arguably better have needed to make large changes (like introducing garbage collection) in order to get enough functional distance from C not to be swamped by it.

This history is worth recalling and understanding because C shows us how powerful a clean, minimalist design can be. If Thompson and Ritchie had been less wise, they would have designed a language that did much more, relied on stronger assumptions, never ported satisfactorily off its original hardware platform, and withered away as the world changed out from under it. Instead, C has flourished — and the example Thompson and Ritchie set has influenced the style of Unix development ever since. As the writer, adventurer, artist, and aeronautical engineer Antoine de Saint-Exupéry once put it, writing about the design of airplanes: «La perfection est atteinte non quand il ne reste rien à ajouter, mais quand il ne reste rien à enlever». ("Perfection is attained not when there is nothing more to add, but when there is nothing more to remove".)

Therefore:

- Stick to plain C.

  C enables us to build data structures for storing sparse matrices, solver information, etc. in ways that Fortran simply does not allow. ANSI C is a complete standard that all modern C compilers support. The language is identical on all machines. C++ is still evolving and compilers on different machines are not identical. Using C function pointers to provide data encapsulation and polymorphism allows us to get many of the advantages of C++

without using such a large and more complicated language.

PETSc FAQs
https://www.mcs.anl.gov/petsc/documentation/faq.html#why-c

- Keep C++ away.

  C++ is a horrible language. It's made more horrible by the fact that a lot of substandard programmers use it, to the point where it's much much easier to generate total and utter crap with it. Quite frankly, even if the choice of C were to do *nothing* but keep the C++ programmers out, that in itself would be a huge reason to use C.

  Linus Torvalds about why Git is written in C
  http://harmful.cat-v.org/software/c++/linus

  Libraries written in C++ are allowed as long as they provide C wrappers and do not meddle with stuff we do not need nor want.

- Keep Fortran even farther away. It is allowed only for existing libraries dating from ancient well-tested and optimized code, but for nothing else.
- Modern high-level languages like Python or Julia are targets and not sources of FeenoX.
- For documentation and comments within the code, American English should be used.

3.3 Programming IDEs

3.3.1 Netbeans

To work on FeenoX using NetBeans, first make sure you have the C/C++ module working. If you don't, do this:

> In Netbeans go to Tools->Plugins->Settings. Entry NetBeans 8.2 Plugin Portal is already present. Click the checkbox next to this entry. Switch to Available Plugins tab, click Check for Newest. C/C++ is now on the list.

Then create a new C/C++ project File->New Project... choose C/C++ Project with Existing Sources click Next, browse to find the root directory of your local FeenoX sources. It is important to have the `configure` script already created there, so make sure you have run `./autogen.sh`. Make sure the configuration mode is set to Automatic and that NetBeans detected the configure script by adding "using configure" next to "Automatic." Click to create the project. This will create a directory `nbproject` which will be ignored by `.gitignore`.

3.4 Makefiles

- FeenoX uses the GNU Autools (i.e. Autoconf and Automake).
- If you really feel that you have to use CMake for your contributions, feel free to do so (Unix rule of diversity) but make sure that at the end of the day `./configure && make` still works.

3.5 Test suite

- The directory `tests` contains the test suite with shell scripts that return appropriate errorlevels according to Automake's generic test scripts. In a nutshell:

When no test protocol is in use, an exit status of 0 from a test script will denote a success, an exit status of 77 a skipped test, an exit status of 99 a hard error, and any other exit status will denote a failure.

- If you add a new feature please also write a test script not just to check your feature works but to prevent further changes by other people (or even by yourself) from breaking the feature.
- Try to make the check results meaningful and not just check for random results. Try to create tests cases with analytical solution or use benchmarks with known results.
- We will eventually add some code coverage tools to have as most lines covered by at least one test.

3.6 Coding style

- K&R 1TBS with no tabs and two spaces per indent https://en.wikipedia.org/wiki/Indentation_style#Variant:_1TBS_(OTBS)

```
void checknegative(x) {
  if (x < 0) {
    puts("Negative");
  } else {
    nonnegative(x);
  }
}
```

- Do not worry about long lines. Only wrap lines when it makes sense to from a logic point of view not because “it looks bad on screen” because “screen” can be anything from a mobile phone to a desktop with many 24” LED monitors.
- Make sure you understand and follow the 17 rules of Unix philosophy https://en.wikipedia.org/wiki/Unix_philosophy

1. modularity
2. clarity
3. composition
4. separation
5. simplicity
6. parsimony
7. transparency
8. robustness
9. representation
10. least surprise
11. silence
12. repair
13. economy
14. generation
15. optimization
16. diversity
17. extensibility

3.7 Virtual methods

Even though we use C, we can still have some "virtual methods" by using function pointers. So in a FEM formulation, the routines that build the stiffness matrix are mostly independent of the problem type (for the same family, say elliptic problems) but in the integrands. So a good practice is to have a common set of routines that loop over elements and over gauss points and then at the inner loop a virtual method is called which depends on the particular problem (thermal, mechanical, etc). This is accomplished by using function pointers so all problems have to provide more or less the same sets of routines: initialization, integrands, BCs, etc. See the basic problems to see how this idea works.

Reading and writing mesh/post-processing files work the same way. Each format has to provide a virtual reader/writer method.

3.8 Memory management

- Check all `malloc()` calls for `NULL`. You can use the `feenox_check_alloc()/feenox_check_alloc_null()` macros.
- Use the macro `feenox_free()` instead of plain `free()`. The former explicitly makes the pointer equal to `NULL` after freeing it, which is handy.
- Use `valgrind` to check for invalid memory access and leaks

  > Memory analysis tools such as valgrind can be useful, but don't complicate a program merely to avoid their false alarms. For example, if memory is used until just before a process exits, don't free it simply to silence such a tool.
  >
  > GNU Coding Standards
  > https://www.gnu.org/prep/standards/html_node/Memory-Usage.html

  Mind that FeenoX might be used in parametric or minimization mode which would re-allocate some stuff so make sure you know which alarms you ignore.

3.9 Naming conventions

- Use snake case such as in `this_is_a_long_name`.
- All functions ought to start with `feenox_`. This is the small price we need to pay in order to keep a magnificent beast like C++ away from our lives (those who can). The names of functions should go from general to particular such as `feenox_expression_parse()` and `feenox_expression_eval()` (and not `feenox_parse_expression()`/`feenox_eval_expression()`) so all function related with expressions can be easily found. There are exceptions, like functions which do similar tasks such as `feenox_add_assignemnt()` and `feenox_add_instructions()`. Here the `add` part is the common one.
- In a similar way, "virtual" methods should add the particularity at the end, like
  - `feenox_build_element_volumetric_gauss_point()`—the function pointer which is invoked in the code
  - `feenox_build_element_volumetric_gauss_point_thermal()`—the particular function the pointer will point to and which is actually called

3.10 Comments

- Use single-line comments `//` to add comments to the code so we can easily *comment out* certain parts of code using multi-line comments `/*`—`*/` while developing new features.

- Explain any non-trivial block or flag that needs to be set. Example

```
switch (c = fgetc(file_ptr)) {
  case '"':
    // if there's an escaped quote, we take away the escape char
    // and put a magic marker 0x1e, afterwards in get_next_token()
    // we change back the 0x1e with the unescaped quote
    feenox_parser.line[i++] = 0x1e;
```

- Focus on the why and not on the how, except for very complex loops.
- All nice-to-have things that are welcome to be done should be written as

```
// TODO: allow refreshing file data before each transit step
```

- Features that might or might not be added should be written as questions

```
// TODO: should we allow unquoted names in the $PhysicalNames section?
```

3.11 Indices in PDEs

Index	Loop over...
`i`	elements or cells
`j`	nodes, either global or local. If there are loops which need both, use `j_local` and `j_global`.
`k`	dummy index
`d`	**d**imensions
`g`	degree of freedoms (from **g**roups of energy actually)
`q`	Gauss (**q**uadrature) points

3.12 Git workflow

- Of course, use `git`. This may seem obvious but predecessors of this project have used Subversion, Bazaar and Mercurial in that order. I have myself used CVS (in the past century).
- Only team members are allowed to commit directly to the main branch.
- All contributions ought to come from pull/merge requests either from forked repositories and/or non-main branches.
- Issues can be opened forked for
  - bug reporting
  - feature requests
  - development discussion
  - general questions (installation, usage, etc)

3.13 Standards

- Try to adhere to POSIX as much as possible. Eventually all operating systems will adopt it.

3.14 Mentioning other libraries, programs, codes, etc.

- Try no to mention any piece of software which is not free and open source.

3.15 Documentation

- See the README in the `doc` directory.
- TLDR;
  1. Use Pandoc-flavored Markdown as the main source.
  2. Knock yourself out with LaTeX math.
  3. Bitmaps are off the table for figures.

4 What we program in FeenoX

- The features described in the SRS.

5 Code of conduct

- See the code of conduct.

6 Release plans

- v1.x first stable release meaning:
  - usable features
  - full documentation matching the code features (PhD thesis)
- v2.x further features and improvements