SMARDDA/DEVELOP

2017-4

Generated by Doxygen 1.8.13

Chapter 1

Developer Guide

To product robust software for reactor design using Fortran, the Fortran programming style guide CCFE-R(15)34 is recommended, see https://scientific-publications.ukaea.uk/wp-content/uploads/CCFE-R-1534.pdf However there is lacking from this guide:

1. References to the literature, to make the point that the Guide is a distillation of ideas from a range of sources, see Sources
2. A list of the objects (classes) already available in the recommended style, see Available Classes for an indication.
3. Anything but the tersest instructions as to how the objects (classes) might be combined in a computational physics project, see Use in Computational Physics
4. Development tools available, see Development tools available

1.1 Sources

Principally web-sites from major national labs and other government bodies such as NASA and national meteorological offices. As a result there is close agreement with the ParaFEM Coding STandard, since this is based on 'European Standards for Writing andDocumenting Exchangeable Fortran 90 Code', https://www.reading.ac.uk/physicsnet/units/3/3phss/F90Style.pdf There are also two books, by Markus https://doi.org/10.1017/CBO9781139084796 and by Clerman and Spector https://doi.org/10.1017/CBO9781139027687 on programming in object-oriented Fortran that were considered at least in part in the production of the guide. Reference sources are the Intel Fortran manuals, available to licensed users of ifort, and "Modern Fortran Explained" by Metcalf et al.
https://global.oup.com/academic/product/modern-fortran-explained-9780199601424?cc=gb&lan

1.2 Available Classes

Apart from SMARDDA, see
http://arxiv.org/abs/1403.6750 http://arxiv.org/abs/1403.7142
the main unclassified development governed by the guide has been for the FISPACT-II activation analysis software, specifically the rate equation, including set up of the matrix elements and analysis of results, notably path analysis. See http://www.fispact.com

Objects have also been produced to solve the problem of converting MCNP particle output (WSSA format) into a form suitable for input to the determinstc neutron transport code Attila, see Arter and Loughlin https://doi.org/10.1016/j.fusengdes.2008.11.051 In addition, studies have been made of, and coding started for, both a replacement for the locally written but now obsolescent MAGINT code (magnetic field defined by circuit elements) and a time dependent 1-D edge transport code, cf. SOLF1D.

1.3 Use in Computational Physics

1.3.1 Advantages of object-oriented Fortran, bash and dialog

The first point to establish is the advantages of object-oriented Fortran. Computational speed would seem to be the main one, given that many libraries in other languages are simply wrappers for FORTRAN that in some cases dates back to the mid-20th Century. The other plus is the longevity of the language in that well-written FORTRAN from so many years ago is still portable to machines with few or no changes. FORTRAN code is, even in the latest standards, reasonably comprehensible to scientifically trained people with little or no coding background.

The advantages of combining object-oriented Fortran with the bash shell, is that proficiency in bash or one of the similar shells is generally found to be useful, if not essential, by Linux users. bash is also very stable, has not significantly evolved for a decade as of 2017, and there is the book by Newham and Rosenblatt, Learning the bash shell 3rd Edition from oreilly.com. Using bash also gives a natural path to the production of the scripts which are typically needed in any scientific project to perform parameter scans.

The developer pack will also include the capability to generate simple dialog-based GUIs for each of the objects, which was helpful in the production of the ismiter GUI for setting up simple geometry and calculations for smiter, see ∼/smardda/smiter/int Again bash dialog should be familiar to Linux users as the interface for system updates and is also stable.

1.3.2 Object-oriented Fortran

The object-oriented approach is central to the efficient solution of the age-old problem of how to get different codes to communicate. Generally it only works if all the code is object-oriented.

Figure 1.1 Object-oriented data flow

The figure shows how in essence an object-oriented physics code should work. The main assumption is that the problem is reducible to a sequence of matrix manipulations, cf. matlab(TM) as "matrix laboratory". Each object contributes to the coefficients of the matrix, which itself becomes an object with associated methods for its inversion/solution.

1.3.3 Hierarchical Objects

The rewriting of nucode shows how a larger piece of software might be developed based on the sample code presented in CCFE-R(15)34, which does not address the problem of a hierarchy of objects. It is natural for the main program to be called nucode, source in nucode.f90 and for the main object/class to be defined in nucode_h.f90 and the associated methods as subroutines in module nucode_m.f90.

The split adopted in nucode.f90 is first, to initialise the control structure (ncontrol_h, methods in ncontrol_m.f90), which of course now consists of a hierarchy of control structures, one for each object forming nucode_h (referred to as a sub-object). Each control structure, typically qnumerics_h some q, is initialised as one or more namelists in a single input file, if ncontrol_read calls structure_readcon, making use of the optional unit number argument. For eaxmple, ncontrol_read calls nucode_readcon, odes_readcon and fmesh_readcon. There is also a routine ncontrol_fix which might be used e.g. to enforce consistency on the various control structures.

With the control structures successfully input, by convention the main program now calls routines to read in the data files specified in program control structure numerics_h, object by object. Each object the input of which is completed

in this way, should have methods/subroutines that separately open the file, read it and close it (though this has not been strictly observed in the past).

The next phase is to complete the initialisation of the top-level nucode object. Ideally nucode_init should call object↩ _init for each sub-object. It also initialises objects for which do not need a separate control structure, in this case poweltn and beam, then it performs one-off transformations of itself and sub-objects, here quantising the coordinates and scaling the magnetic field to be used in the particle advance.

Once the nucode object is defined (instantiated), the main calculation is coordinated by nucode_compute and its results analysed by nucode_calc and nucode_stat. nucode_compute is at the head of a deep nesting of routines, nucode_compute -> nucode_move -> beam_move -> pclist_init and pcle_move. The pclist object is a list of particles (posnode_t objects) which is generated dynamically by nucode_compute and so cannot be initialised earlier. Note that for historical reasons a particle is not a pcle_t object even though it is defined in file pcle_h.f90.

Once the computation is completed, there are separate controls for each file to output from nucode.f90. To help write .vtk format files, the vfile_h object and methods are available, and similarly for gnuplot format files, the gfile_h and methods should be used. However, each sub-object will have to have its own _writev or _writeg subroutine to perform its own output. By convention, a .out file is also produced. This may contain nothing of importance, its main function's being to act as a lock file and prevent accidental overwriting of output from previous runs with the same input/name.

Lastly, it is good practice to call routines to delete, ie. deallocate storage for, nucode and its sub-objects, and close any files that might still be open from nucode_compute.

1.4 Development tools available

The level of sophistication of these tools is very variable, some are reasonably polised scripts, others are just collections of useful edits that will do a lot of the work but still need manual intervention to finish it.

The example source associated with CCFE-R(15)34 is in smardda sub-repository qprog, mirroring https↩ ://github.com/wayne-arter/smardda-qprog In fact, each example file is in three parts, a header containing possibly legally valid stuff without technical value, a description of the module and thirdly its contents. The first header part need not appear in most cases, and the second part should be in a separate file object.doc, so that the developer can start editing at line 1.