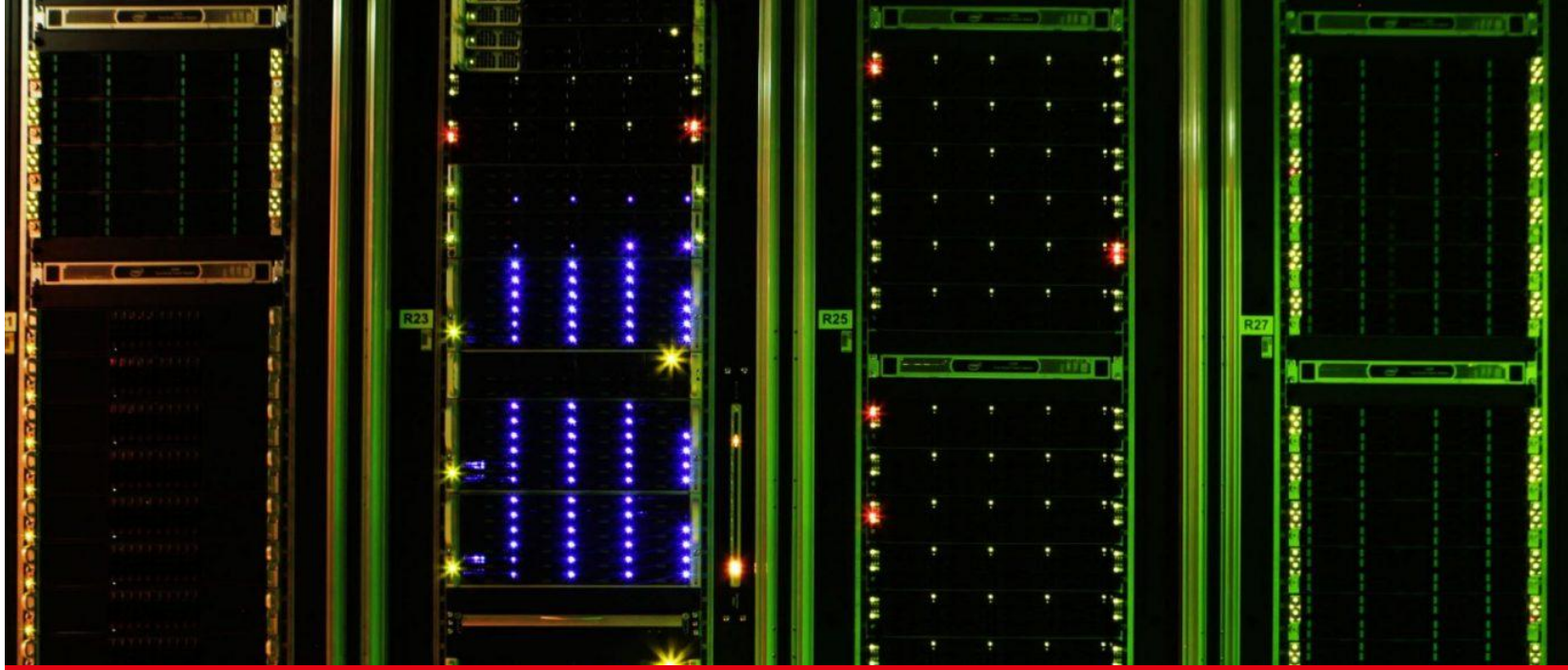
EPFL

EUROfusion



From SPEC to SPECTRE: breathing Pythonic life into a legacy Fortran code

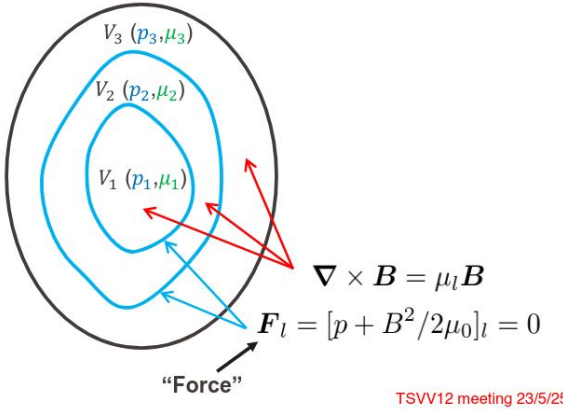
E. Lanti, E. Balkovic, C. Smiet, J. Loizu

3rd Annual Meeting of EUROfusion HPC ACHs
Tue. 25th of November 2025, Lausanne

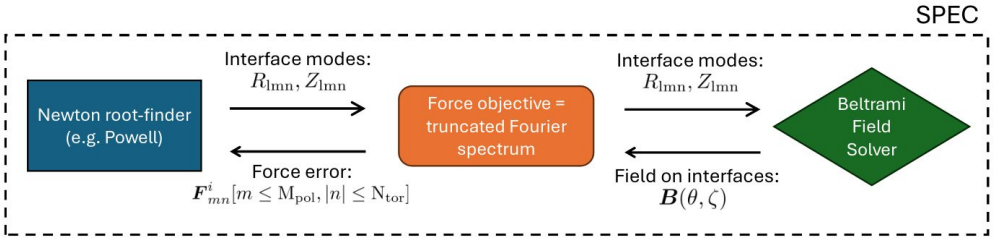
SPEC code

Computing magnetic equilibria

- Compute 3D MHD equilibria using MRxMHD
- Volume is subdivided into regions with $\nabla p=0$
- Interfaces between regions are force-free



- SPEC does this in two main steps:
  - Field computation
  - Optimization of the surfaces

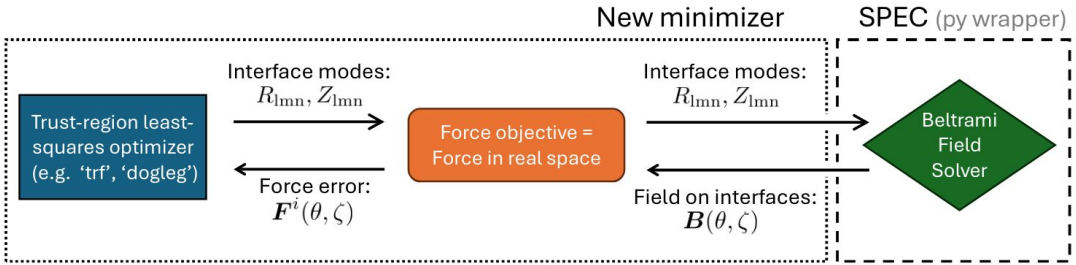


- Doesn't converge for highly shaped equilibria with many interfaces

SPEC code

Addressing the convergence problem

- The SPC team developed a new minimizer solving the convergence problem
  - Field is computed by SPEC (Fortran + MPI + OpenMP)
  - Optimization is done by an home-made Python package interfaced to SPEC



- SPEC became a black-box from which we scavenge parts to port in the new code

From SPEC to SPECTRE

- Fortran codebase modernization and standardization
- Build system re-writing
- Python bindings
- Code quality
- CI containerization
- Open Source licensing

Modernization of the Fortran codebase

- Make the code more maintainable, testable and extensible
- For this the main objectives were:
  - Conform to the standard as much as possible
  - Remove dead and unused code (using code coverage)
  - Encapsulate related code into modules and set correct visibility
  - Reduce use of global variables
  - Impose coding conventions and guidelines

```
REAL    :: Tmanual = 0.0, manualT = 0.0
REAL    :: Trzaxis = 0.0, rzaxisT = 0.0
REAL    :: Tpackxi = 0.0, packxiT = 0.0
REAL    :: Tvolume = 0.0, volumeT = 0.0
REAL    :: Tcoords = 0.0, coordsT = 0.0
REAL    :: Tbasefn = 0.0, basefnT = 0.0
REAL    :: Tmemory = 0.0, memoryT = 0.0
REAL    :: Tmetrix = 0.0, metrixT = 0.0
REAL    :: Tma00aa = 0.0, ma00aaT = 0.0
REAL    :: Tmatrix = 0.0, matrixT = 0.0
REAL    :: Tspsmat = 0.0, spsmatT = 0.0
REAL    :: Tspsint = 0.0, spsintT = 0.0
REAL    :: Tmp00ac = 0.0, mp00acT = 0.0
REAL    :: Tma02aa = 0.0, ma02aaT = 0.0
REAL    :: Tpackab = 0.0, packabT = 0.0
REAL    :: Ttr00ab = 0.0, tr00abT = 0.0
REAL    :: Tcurent = 0.0, curentT = 0.0
REAL    :: Tdf00ab = 0.0, df00abT = 0.0
REAL    :: Tlforce = 0.0, lforceT = 0.0
REAL    :: Tintghs = 0.0, intghsT = 0.0
```

```
numrec.f90  pp00ab.f90  spsint.f90
packab.f90  preset.f90  spsmat.f90
packxi.f90  ra00aa.f90  stzxyz.f90
pc00aa.f90  rksuite.f   tr00ab.f90
pc00ab.f90  rzaxis.f90  volume.f90
pp00aa.f90  sphdf5.f90  wa00aa.f90
```

Removing M4 macros

- The SPEC code heavily relies on macros defined in M4 language
- Complexify the building process
- They make the Fortran code difficult to read
- Most of those macros could be written as Fortran subroutines

Removing M4 macros

- The SPEC code heavily relies on macros defined in M4 language
- Complexify the building process
- They make the Fortran code difficult to read
- Most of those macros could be written as Fortran subroutines

```
INTEGER             :: vvol, Ndofgl, iflag, cpu_send_one, cpu_send_two, ll, NN, ideriv, iocons
INTEGER             :: status(MPI_STATUS_SIZE), request1, request2
REAL                :: Fvec(1:Ndofgl), x(1:Mvol-1), Bt00(1:Mvol, 0:1, -1:2), ldItGp(0:1, -1:2)
LOGICAL             :: LComputeDerivatives
INTEGER             :: deriv, Lcurvature
```

Removing M4 macros

- The SPEC code heavily relies on macros defined in M4 language
- Complexify the building process
- They make the Fortran code difficult to read
- Most of those macros could be written as Fortran subroutines

```
INTEGER             :: vvol, Ndofgl, iflag, cpu_send_one, cpu_send_two, ll, NN, ideriv, iocons
INTEGER             :: status(MPI_STATUS_SIZE), request1, request2
REAL                :: Fvec(1:Ndofgl), x(1:Mvol-1), Bt00(1:Mvol, 0:1, -1:2), ldItGp(0:1, -1:2)
LOGICAL             :: LComputeDerivatives
INTEGER             :: deriv, Lcurvature
```

```
m4_define(INTEGER,integer)m4_dnl ; can put comments here;
m4_define(REAL,real(8))m4_dnl ; can put comments here;
```

Removing M4 macros

- The SPEC code heavily relies on macros defined in M4 language
- Complexify the building process
- They make the Fortran code difficult to read
- Most of those macros could be written as Fortran subroutines

```
INTEGER             :: vvol, Ndofgl, iflag, cpu_send_one, cpu_send_two, ll, NN, ideriv, iocons
INTEGER             :: status(MPI_STATUS_SIZE), request1, request2
REAL                :: Fvec(1:Ndofgl), x(1:Mvol-1), Bt00(1:Mvol, 0:1, -1:2), ldItGp(0:1, -1:2)
LOGICAL             :: LComputeDerivatives
INTEGER             :: deriv, Lcurvature
```

```
m4_define(INTEGER,integer)m4_dnl ; can put comments here;
m4_define(REAL,real(8))m4_dnl ; can put comments here;
```

```
! (en/dis)able HDF5 internal error messages; sphdf5 has its own error messages coming from the macros
H5CALL( sphdf5, h5eset_auto_f, (internalHdf5Msg, hdfier), __FILE__, __LINE__)

! Create the file
H5CALL( sphdf5, h5fcreate_f, (trim(ext)//".sp.h5", H5F_ACC_TRUNC_F, file_id, hdfier ), __FILE__, __LINE__ )
```

Managing precision using kinds

- By default, Fortran `real` numbers are of "single precision"
- Different techniques to impose real "double precision"
  - Use compiler flag
  - Use Fortran `double precision` type
  - Use kinds !
- Kind parameters are integers specifying which type representation to use (how bits are mapped to numbers)
- For reals, one could define a kind representing IEEE-754 double precision using:

```
!> Kind value for 64bits IEEE-754 (~15 digits, 10^±307)
integer, parameter :: dp = selected_real_kind(15, 307)
```

- Could also use `real64` from `iso_fortran_env`, but it is less portable[1]

[1] https://stevelionel.com/drfortran/2017/03/27/doctor-fortran-in-it-takes-all-kinds/

Managing precision using kinds

Quiz time

```
1   program test_kinds
2       implicit none
3
4       !> Kind value for 64bits IEEE-754 floats (~15 digits, 10^±307)
5       integer, parameter :: dp = selected_real_kind(15, 307)
6
7       real :: pi_sp_sp = 3.1415926535897932385
8       real(dp) :: pi_dp_sp = 3.1415926535897932385
9       real(dp) :: pi_dp_dp = 3.1415926535897932385_dp
10
11      print*, "pi_sp_sp", pi_sp_sp
12      print*, "pi_dp_sp", pi_dp_sp
13      print*, "pi_dp_dp", pi_dp_dp
14  end program test_kinds
15
```

Managing precision using kinds

Quiz time

```
1   program test_kinds
2       implicit none
3
4       !> Kind value for 64bits IEEE-754 floats (~15 digits, 10^±307)
5       integer, parameter :: dp = selected_real_kind(15, 307)
6
7       real :: pi_sp_sp = 3.1415926535897932385
8       real(dp) :: pi_dp_sp = 3.1415926535897932385
9       real(dp) :: pi_dp_dp = 3.1415926535897932385_dp
10
11      print*, "pi_sp_sp", pi_sp_sp ! 3.14159274
12      print*, "pi_dp_sp", pi_dp_sp
13      print*, "pi_dp_dp", pi_dp_dp
14  end program test_kinds
15
```

Managing precision using kinds

Quiz time

```
1   program test_kinds
2       implicit none
3
4       !> Kind value for 64bits IEEE-754 floats (~15 digits, 10^±307)
5       integer, parameter :: dp = selected_real_kind(15, 307)
6
7       real :: pi_sp_sp = 3.1415926535897932385
8       real(dp) :: pi_dp_sp = 3.1415926535897932385
9       real(dp) :: pi_dp_dp = 3.1415926535897932385_dp
10
11      print*, "pi_sp_sp", pi_sp_sp ! 3.14159274
12      print*, "pi_dp_sp", pi_dp_sp ! 3.1415927410125732
13      print*, "pi_dp_dp", pi_dp_dp
14  end program test_kinds
15
```

Managing precision using kinds

Quiz time

```
1   program test_kinds
2       implicit none
3
4       !> Kind value for 64bits IEEE-754 floats (~15 digits, 10^±307)
5       integer, parameter :: dp = selected_real_kind(15, 307)
6
7       real :: pi_sp_sp = 3.1415926535897932385
8       real(dp) :: pi_dp_sp = 3.1415926535897932385
9       real(dp) :: pi_dp_dp = 3.1415926535897932385_dp
10
11      print*, "pi_sp_sp", pi_sp_sp ! 3.14159274
12      print*, "pi_dp_sp", pi_dp_sp ! 3.1415927410125732
13      print*, "pi_dp_dp", pi_dp_dp ! 3.1415926535897931
14  end program test_kinds
15
```

Managing precision using kinds

Quiz time

```
1   program test_kinds
2       implicit none
3
4       !> Kind value for 64bits IEEE-754 floats (~15 digits, 10^±307)
5       integer, parameter :: dp = selected_real_kind(15, 307)
6
7       real :: pi_sp_sp = 3.1415926535897932385
8       real(dp) :: pi_dp_sp = 3.1415926535897932385
9       real(dp) :: pi_dp_dp = 3.1415926535897932385_dp
10
11      print*, "pi_sp_sp", pi_sp_sp ! 3.14159274
12      print*, "pi_dp_sp", pi_dp_sp ! 3.1415927410125732
13      print*, "pi_dp_dp", pi_dp_dp ! 3.1415926535897931
14  end program test_kinds
15
```

- In addition to adding kind parameters, one must also take care of literals, and functions, *e.g.* `cmplx`
  - Difficulty: literals can be written 1.0, 1.0E0, 1.0D0, 1D0, 1E0, 1.D0, 1.E0, .1, .1D0, *etc.*

Cleaning the build configuration

- Remove outdated Makefile configuration
- Rewrite CMake configuration and adhere to modern CMake
  - Reduce complexity (450 lines to 150 lines)

Cleaning the build configuration

- Remove outdated Makefile configuration
- Rewrite CMake configuration and adhere to modern CMake
  - Reduce complexity (450 lines to 150 lines)

- Use of CMake presets
  - Store project configuration in version-controlled JSON files, “CMakePresests.json”
  - Must-have for CI build, IDEs, *etc*.
  - Simplify user experience
  - Separate generic configuration from machine/environment specific

```
cmake -B build \
    -DCMAKE_BUILD_TYPE=Release \
    -DENABLE_MPI_F08=ON \
    -DUSE_OPENMP=OFF \
    -DUSE_OPENACC=ON
```

```
> cmake --preset release-mpif08-openacc
Preset CMake variables:

  CMAKE_BUILD_TYPE="Release"
  USE_MPI_F08="ON"
  ORB5_OPENMP="ON"
  [...]
```

Interfacing Fortran with Python

- There are three main ways of interfacing Fortran and Python
  - Using ctypes to load a shared object
  - Writing a Fortran <-> C interface and use PyBind
  - Using f2py and f90wrap
- Later is chosen because less boilerplate code
  - f90wrap generates simpler Fortran interfaces for f2py and a Python interface module
  - f2py generates the Fortran <-> C interface
- Scikit-build-core is used as a build backend for CMake <-> Python

- Limitations of f90wrap
  - project is maintained by one person and documentation is gappy
  - Not working as advertised (issue opened with solution since August 6th)
  - Downcase everything

Improving code quality

- Various tools and workflow have been used to improve code quality
- Fortran
  - Fortitude: an open source Fortran linter built upon Ruff
  - Codee formatter: a free commercial code formatter
- Python
  - Ruff: (extremely fast) linter and code formatter
- Treatment of the warnings from the different tools
- Coding guidelines
- Code review

CI setup

- New CI setup currently being implemented
  - Build, test, code quality and doc generation
- Docker image containing an extensible software stack to compile SPECTRE

CI setup

- New CI setup currently being implemented
  - Build, test, code quality and doc generation
- Docker image containing an extensible software stack to compile SPECTRE

```yaml
spack:
  definitions:
    - core-packages:
      - cmake
      - lmod
    - packages:
      - hdf5+hl+fortran+mpi
      - openmpi
      - openblas
      - fftw
      - python@3.11
      - py-numpy
      - py-mpi4py

    - compilers:
      - gcc@13.2.0

  specs:
    - matrix:
      - [$compilers]
    - matrix:
      - [$core-packages]
      - ['%gcc@13.2.0']
    - matrix:
      - [$packages]
      - [$%compilers]
```

- Easily generate software stack using Spack
  - Core packages: built once with default compiler
  - Packages: built by each compiler
  - Compilers
- Use also by users for reproducible build environment

Spack

Licensing

- SPEC was licensed under GPLv3
  - Can be problematic, especially for collaboration with private companies
- Original authors accepted to switch to MIT license
- SPECTRE code available shortly on GitLab.com:

  https://gitlab.com/spectre-eq/spectre

Conclusions

- The new SPECTRE code is about to be released under MIT license
- It is born out of the original SPEC

- Fortran code has been refactored and standardized
- It is interfaced to the new Python minimizer
- Everything is build consistently using CMake and scikit-build-core

- Code quality has been improved using good practices and tools:
  - Linter and code formatter
  - Coding guidelines and conventions
  - Systematic code review

- The community is eager to use this new SPECTRE code