Daniel Forstenlechner

Unit tests in Fortran using the example of GORILLA

Bachelor's Thesis
to achieve the university degree of
Bachelor of Science
Bachelor's degree programme: Physik

submitted to

Graz University of Technology

Supervisor: Dipl.-Ing. Dr.techn. Michael Eder, BSc
Co-Supervisor: Ass.Prof. Dipl.-Ing. Dr.rer.nat. Christopher Albert, BSc

Institute of Theoretical and Computational Physics
Head: Univ.-Prof. Dipl.-Phys. Dr.rer.nat. Wolfgang von der Linden

Graz, 03 2022

Abstract

This thesis shows an exemplary approach on modifying an existing Fortran project to include a unit test framework. The project chosen to demonstrate the approach is GORILLA: Guiding-center ORbit Integration with Local Linearization Approach. Furthermore, an example unit test is implemented, which acts as template for additional unit tests. The amount of code which is tested by the unit tests is displayed in a code coverage report, and the build of the application and the tests are automatized. The various tools which are needed to perform the tasks are described. Using the steps shown in this thesis, it is now possible to implement unit tests on similar projects with ease.

Contents

1 Bachelor's thesis' objective

The objective of this thesis is to modernize the build of GORILLA[1], a program to compute guiding-center orbits for charged particles, and to implement a Fortran unit test framework into the build. Furthermore, an example unit test shall be written and the code coverage shall be determined. The example unit test acts as a template for further unit tests. Therefore, the implementation of additional unit tests via the unit test framework and the build tool should be as simple as possible. At last, the build of GORILLA and the generation of the code coverage reports are automatized with the help of GitHub workflows.
The main reason for doing the above-mentioned steps is to improve GORILLA and to have automated tests to make sure the code is running correctly. Additionally, GORILLA should act as an example project, which can then be used to perform similar steps on other projects of the plasma physics group of the Institute of Theoretical and Computational Physics at the TU Graz[2]. For example, SIMPLE[3], a program to compute statistical losses of guiding-center orbits for particles, shall be improved in the same way.
The thesis is structured as follows: Section 2 is giving an overview on GORILLA. In section 3 basics on unit tests and unit test frameworks are provided. Afterwards, the used tools are described in section 4. The modernization of the build of GORILLA, the implementation of the unit test framework and the unit tests are described in section 5.1 to 5.3. The results are then used in section 5.4 to generate the code coverage report. Section 5.5 describes the GitHub workflow. The results of the thesis are summarized and discussed in section 6.
During the execution of the tasks of the bachelor's thesis, two sample projects were created, which are available on GitHub[4, 5]. These projects are the prototypes for the implementation of unit tests in GORILLA and are not further discussed in this thesis.

2 Introduction on GORILLA

The following part can be found in the paper "GORILLA: Guiding-center ORbit Integration with Local Linearization Approach", which was submitted to JOSS, the Journal of Open Source Software. The paper was rejected and will be resubmitted after improving different parts of GORILLA, including automatized tests, which are described in this thesis. The author of this thesis is a co-author of this paper. To see the raw text of the paper, take a look at the file on GitHub[6].

Introduction

"GORILLA is a Fortran code that computes guiding-center orbits for charged particles of given mass, charge and energy in toroidal fusion devices with three-dimensional field geometry. Conventional methods for integrating the guiding-center equations utilize high order interpolation of the electromagnetic field in space. In GORILLA, a special linear interpolation employing a spatial mesh is used for the discretization of the electromagnetic field. This leads to locally linear equations of motion with piecewise constant coefficients. As shown by M. Eder et al. (2020)[7], this local linearization approach retains the Hamiltonian structure of the guiding-center equations. For practical purposes this means that the total energy, the magnetic moment and the phase space volume are conserved. Furthermore, the approach reduces computational effort and sensitivity to noise in the electromagnetic field. In GORILLA guiding-center orbits are computed without taking into account collisions in-between particles."[1]

Statement of need

"GORILLA is designed to be used by researchers in scientific plasma physics simulations in the field of magnetic confinement fusion. In such complex simulations a simple interface for the efficient integration of the guiding-center equations is needed. Specifically, the initial condition in five-dimensional phase space is provided (i.e. guiding-center position, parallel and perpendicular velocity) and the main interest is in the condition after a prescribed time step while the integration process itself is irrelevant. Such a pure "orbit time step routine" acting as an interface with a plasma physics simulation is provided. The integration process itself, however, can also be of great interest and a program allowing the detailed analysis of guidingcenter orbits, the time evolution of their respective invariants of motion and Poincaré plots is thus also provided. GORILLA has already been used by M. Eder et al. (2020)[7] for the application of collisionless guiding-center orbits in an axisymmetric tokamak and a realistic three-dimensional stellarator configuration. There, the code demonstrated stable long-term orbit dynamics conserving invariants. Further, in the same publication, GORILLA was applied to the Monte Carlo evaluation of transport coefficients. There, the computational efficiency of GORILLA was shown to be an order of magnitude higher than with a standard fourth order Runge–Kutta integrator. Currently, GORILLA is part of the "EUROfusion Theory, Simulation, Validation and Verification Task for Impurity Sources, Transport, and Screening" where it is tested for the kinetic modelling of the impurity ion component. The source code for GORILLA has been archived on Zenodo with the linked DOI: (Michael Eder et al., 2021[8])"[1]

3 Fundamentals on Unit Tests

This chapter provides information on unit tests and unit test frameworks. The following contents are based on chapter 1 of the book "Unit Test Frameworks"[9]. *This chapter is not considered to be an independent work of the author, but outlines a brief review of the theoretical background for the following chapters.*

Unit tests are a standard method in the quality assurance process of a software. They are designed for testing particular behaviour of a single unit of code of the production code of an application. The simplest form of a unit test is a few lines of throwaway code that, for example, call a subroutine and check whether the returning value is the expected one. This approach is not suitable for lager projects, because it does clutter up the production code and increases the size of the compiled application. To avoid this, unit test frameworks are used. Unit test frameworks are software tools, which provide methods to write tests, execute tests, collect the results and return it to the user. Unit tests are not a part of the final application, and the relation between the application and the unit test framework is shown in figure 1.

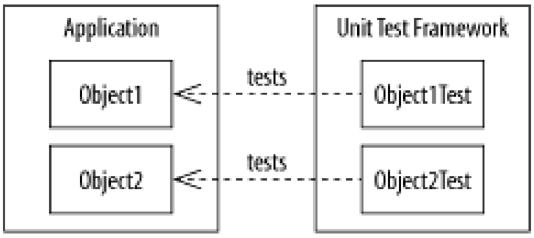


Figure 1: Relation between Unit Test Framework and Application[9, Figure 1-1]

A single unit test usually sets up a scenario, then executes a part of the production code and checks the result or behaviour of it. The scenarios should be as simple as possible and not dependent on results of other tests. To test larger objects, it is useful to start to test the basic functionality and then build up more sophisticated tests. The unit test framework then uses the results from the unit tests and reports them back to the programmer.
Unit tests are usually white box tests due to the fact that the tests can access the internal structure of the code. Apart from that, so-called black box tests exist. These tests are running the application without knowing the internal structure and only check the return code.
Unit tests can be programmer or acceptance tests. Programmer tests test low-level functionalities, e.g. subroutines which do not call other subroutines. Acceptance tests test high-level behaviour, e.g. the behaviour of a subroutine which calls other subroutines.

4 Used Tools and installation

4.1 make

Make[10] is an open source build automation tool. It uses so called makefiles, in which the process how to compile and link source files to executable programs and libraries is described in a distinct script language. The advantage of make is, that the end user does not need to know how the compilation and linking process of a program is done. It is only necessary to execute make to install it on the computer of the end user. Nowadays, the standard version is GNU Make which can be obtained from the GNU Homepage[10] or from different app stores available on UNIX systems (apt[11], snap[12], brew[13]). GNU make is distributed under the GNU General Public License.

4.2 CMake

CMake[14] is an open source development tool for building and testing of software by the company Kitware. It is not a build automation tool like make, but it creates the makefiles for the user. The advantages of CMake are that the CMakeLists are easier to maintain than makefiles and that there is a large documentation[15]. Another advantage of CMake is the so called CTest[16, 17], which collects and executes the tests of the program and writes out the results. The source code of CMake is available on GitHub[18]. To obtain CMake, take a look at the installation guide[19]. CMake is distributed under the OSI-approved BSD 3-clause License.

4.3 pFUnit

Due to the limited usage of Fortran in modern programming, only a few unit test frameworks are available. For an overview, take a look at [20] and [21]. Most of these frameworks are not well documented and are outdated due to a lack of active programmers. The choice for pFUnit[22] was made due to active programmers and community, a simple documentation and installation guide and the availability of demo projects. pFUnit is a unit testing framework for Fortran created by developers from NASA and NGC TASC. The acronym stands for Parallel Fortran Unit Testing Framework. It gives the user a lot of additional functions for Fortran to test code, e.g. assertions which are not part of the standard Fortran language. Furthermore, it works with make and CMake. To install pFUnit see the GitHub page. pFUnit is distributed under the NASA Open Source Agreement for GSC-15,137-1 F-UNIT, also known as pFUnit.

4.4 gcov and lcov

gcov is part of GNU GCC and is distributed under the GNU General Public License. gcov is used to determine the code coverage, that means it tells the programmer which lines of code were executed and how often. However, it does not tell, if the implementation of the code is correct, which is the responsibility of the executed tests. A manual can be found on the GNU GCC homepage[23]. It is automatically installed together with the compiler collection, which is available via the source files[24] or various app stores.
lcov is a graphical front-end for gcov, it is used to visualize the data generated by gcov. lcov is part of the Linux Test Project. To obtain lcov see the GitHub page[25] or the different app stores. lcov is distributed under the GNU General Public License v2.0.

5 Implementation of Unit Tests in GORILLA

5.1 Implementation of CMake

The currently used method to build GORILLA is make. To modernize this, CMake is used (see section 4.2). It is necessary to understand the folder structure of GORILLA to be able to implement CMake. In the ROOT folder are various folders (e.g. DOCUMENTATION, EXAMPLES). The necessary files to build GORILLA are the makefile in the ROOT folder and the Fortran source files in the ROOT/SRC and ROOT/SRC/contrib folder. Below is the directory structure. Relevant files are explicitly mentioned and newly added files and folders for the implementation of CMake, unit tests and additional GitHub workflows are marked blue.

```
<ROOT>
├─ build.sh
├─ build_coverage.sh
├─ CMakeLists.txt
├─ Makefile
├─ SRC
│  ├─ CMakeLists.txt
│  ├─ binsrc.f90
│  ├─ various_functions_mod.f90
│  ├─ <remaining source files>
│  ├─ contrib
│  │  └─ <various external source files>
│  └─ TESTS
│     ├─ CMakeLists.txt
│     ├─ test_binsrc.f90
│     └─ test_various_functions_mod.f90
├─ .github
│  └─ workflows
│     ├─ Mac.yml
│     ├─ Ubuntu.yml
│     ├─ build.yml
│     └─ draft-pdf.yml
└─ BUILD
   └─ <build files>
```

Properties of the Makefile

The next step is to analyze the currently used makefile. The source code is shown below in listing 1. Line 1 sets the variable *FC* for the used compiler, in this case gfortran, which is part of GCC, the GNU Compiler Collection[26]. The next line sets the variable *OPTS* for the compiler and linker options. From line 4 to 11 two variables are created which include the information where to look for specific libraries. The *if* clause is for separating between macOS and other UNIX operating systems. From line 13 through 56 the source files are written to a variable called *SOURCES*. Line 59 and 60 instruct make how to link the executable file test_gorilla_main.x. Line 60 specifically includes the used compiler and linker options. Line 62 to 66 describe how to compile the source files, with the first paragraph describing how to handle the files in the ROOT/SRC/contrib folder and the second paragraph how to handle the files in ROOT/SRC folder. The last few lines are for cleaning up the build.

```
1  FC = gfortran
2  OPTS ?= -J OBJS  -g -fbacktrace -ffpe-trap=zero,overflow,invalid  -fbounds-check -fopenmp
3  
4  UNAME_S := $(shell uname -s)
5  ifeq ($(UNAME_S),Darwin)
6    NCINC ?= -I/opt/local/include
7    NCLIB ?= -L/opt/local/lib -lnetcdff -lnetcdf -llapack
8  else
9    NCINC ?= -I/usr/include
10   NCLIB ?= -lnetcdff -lnetcdf -llapack
11 endif
12 
13 SOURCES = SetWorkingPrecision.f90\
14   Polynomial234RootSolvers.f90 \
15   constants_mod.f90 \
16   tetra_grid_settings_mod.f90 \
17   gorilla_settings_mod.f90 \
18   various_functions_mod.f90 \
19   gorilla_diag_mod.f90 \
20   canonical_coordinates_mod.f90 \
21   nctools_module.f90 \
22   rkf45.f90 \
23   odeint_rkf45.f90 \
24   runge_kutta_mod.f90 \
25   magfie.f90 \
26   chamb_m.f90 \
27   vmecinm_m.f90 \
28   spl_three_to_five_mod.f90 \
29   spline_vmec_data.f90 \
30   new_vmec_allocation_stuff.f90 \
31   binsrc.f90 \
32   field_divB0.f90 \
33   scaling_r_theta.f90\
34   field_line_integration_for_SYNCH.f90 \
35   preload_for_SYNCH.f90 \
36   plag_coeff.f90 \
37   magdata_in_symfluxcoord.f90 \
38   points_2d.f90\
39   circular_mesh.f90\
40   tetra_grid_mod.f90 \
```

```
41   make_grid_rect.f90 \
42   bdivfree.f90 \
43   tetra_physics_mod.f90 \
44   tetra_physics_poly_precomp_mod.f90 \
45   differentiate.f90 \
46   spline5_RZ.f90 \
47   supporting_functions_mod.f90 \
48   pusher_tetra_func_mod.f90 \
49   pusher_tetra_poly.f90 \
50   pusher_tetra_rk.f90 \
51   get_canonical_coordinates.f90 \
52   orbit_timestep_gorilla.f90 \
53   gorilla_plot_mod.f90 \
54   test_gorilla_main.f90
55 
56 OBJS = $(patsubst %.f90,OBJS/%.o,$(SOURCES))
57 
58 
59 test_gorilla_main.x: $(OBJS_CONTRIB) $(OBJS)
60   $(FC) $(OPTS) -o $@ $^ $(NCLIB)
61 
62 OBJS/%.o: SRC/contrib/%.f90
63   $(FC) $(OPTS) -c $^ -o $@ $(NCINC)
64 
65 OBJS/%.o: SRC/%.f90
66   $(FC) $(OPTS) -c $^ -o $@ $(NCINC)
67 
68 .PHONY: clean
69 clean:
70   rm -f OBJS/*
71   rm -f SRC/*.mod
72   rm -f test_gorilla_main.x
```

Listing 1: ROOT/Makefile

CMakeList of the ROOT folder

CMake uses so called CMakeLists[15]. It is possible to use one large CMakeList, but for reasons of maintenance it is useful to write a CMakeList file for every folder. Below is the source code of the CMake file of the ROOT folder in listing 2. Line 1 through 5 are setting the required CMake version with the *cmake_minimum_required* command[27], and the name, version and used programming languages of this project with the *project* command[28]. Line 7 to 11 is similar to line 4 to 11 in the makefile. The *include_directories* command tells CMake which folders to include[29]. In these folders, CMake can search for include files (e.g. LAPACK). The next three lines are for searching for external libraries via the *find_package* command[30]. Line 17 through 20 is for the unit tests and will be discussed later. Line 22 and 23 set the compiler and linker options with the commands *add_compile_options*[31] and *add_link_options*[32]. These compile and link options are valid for the current directory and its subdirectories that are added after this command is invoked. It is possible to add specific options for targets via the *target_compile_options*[33] and *target_link_options*[34] command. Line 25 through 27 add a linker option for macOS operating system. The next few lines are specific for the tests and will be discussed later. The last line tells CMake to look for additional CMakeLists in the SRC folder with the *add_subdirectory* command[35].

```
cmake_minimum_required(VERSION 3.12)

project (GORILLA
  VERSION 1.0.0
  LANGUAGES Fortran)

if(UNIX AND NOT APPLE)
  include_directories(/usr/include)
elseif(APPLE)
  include_directories(/opt/local/include)
endif()

find_package(BLAS REQUIRED)
find_package(LAPACK REQUIRED)
find_package(netCDF REQUIRED)

if ($ENV{GORILLA_COVERAGE} STREQUAL "TRUE")
  find_package(PFUNIT REQUIRED)
  enable_testing()
endif()

add_compile_options(-g -fbacktrace -ffpe-trap=zero,overflow,invalid -fbounds-check -fopenmp)
add_link_options(-g -fbacktrace -ffpe-trap=zero,overflow,invalid -fbounds-check -fopenmp)

if(APPLE)
  add_link_options(-L/opt/local/lib)
endif()

if ($ENV{GORILLA_COVERAGE} STREQUAL "TRUE")
  add_link_options(--coverage)
endif()

add_subdirectory(SRC)
```

Listing 2: ROOT/CMakeList.txt

CMakeList of the SRC folder

The next CMakeList is in the SRC folder. The source code is shown below in listing 3. With the *add_library*[36] command, a library is added to the build, which includes the compiled source files mentioned from row 2 to 43. The result is a file called libGORILLA.a. It is also possible to link it dynamically to get a shared object. Then the result will be libGORILLA.so. Line 45 to 49 are specific for the unit tests and will be discussed later. With the *add_executable*[37] command, an executable is added to the build. The *target_link_libraries*[38] command in line 55 gives CMake the information with which libraries the executable has to get linked. Using the *set_target_properties*[39] command, CMake sets the variable *Fortran_MODULE_DIRECTORY*[40] of the GORILLA library. This variable describes where the compiled source files of the GORILLA library will be placed. In line 60 the folder which includes the source files for compiling the GORILLA library is set via the *target_include_directories*[41] command. The last three lines are for the unit tests and will be discussed later.

```
add_library(GORILLA
  SetWorkingPrecision.f90
  contrib/Polynomial234RootSolvers.f90
  constants_mod.f90
  tetra_grid_settings_mod.f90
  gorilla_settings_mod.f90
  various_functions_mod.f90
  gorilla_diag_mod.f90
  canonical_coordinates_mod.f90
  nctools_module.f90
  contrib/rkf45.f90
  odeint_rkf45.f90
  runge_kutta_mod.f90
  magfie.f90
  chamb_m.f90
  vmecinm_m.f90
  spl_three_to_five_mod.f90
  spline_vmec_data.f90
  new_vmec_allocation_stuff.f90
  binsrc.f90
  field_divB0.f90
  scaling_r_theta.f90
  field_line_integration_for_SYNCH.f90
  preload_for_SYNCH.f90
  plag_coeff.f90
  magdata_in_symfluxcoord.f90
  points_2d.f90
  circular_mesh.f90
  tetra_grid_mod.f90
  make_grid_rect.f90
  bdivfree.f90
  tetra_physics_mod.f90
  tetra_physics_poly_precomp_mod.f90
  differentiate.f90
  spline5_RZ.f90
  supporting_functions_mod.f90
  pusher_tetra_func_mod.f90
  pusher_tetra_poly.f90
  pusher_tetra_rk.f90
  get_canonical_coordinates.f90
  orbit_timestep_gorilla.f90
  gorilla_plot_mod.f90
  )

if ($ENV{GORILLA_COVERAGE} STREQUAL "TRUE")
  target_compile_options(GORILLA
    PRIVATE --coverage
    )
endif()

add_executable(test_gorilla_main.x
  test_gorilla_main.f90
  )

target_link_libraries(test_gorilla_main.x GORILLA netcdff netcdf lapack)

set_target_properties (GORILLA PROPERTIES
```

```
58   Fortran_MODULE_DIRECTORY ${CMAKE_CURRENT_BINARY_DIR})
59
60 target_include_directories(GORILLA PUBLIC ${CMAKE_CURRENT_BINARY_DIR})
61
62 if ($ENV{GORILLA_COVERAGE} STREQUAL "TRUE")
63   add_subdirectory(TESTS)
64 endif()
```

Listing 3: ROOT/SRC/CMakeLists.txt

Build bash script

With these two above described CMakeLists everything is set up to compile GORILLA without unit tests. The steps are described in the build.sh bash script in the ROOT folder shown below in listing 4. The first line is a so called shebang[42]. It tells the operating system how to interpret the file. Afterwards, a check is done, if there is already a BUILD folder, and if this is the case the build folder is removed in line 5. Line 8 only exports a variable, which disables the build of the unit tests in the CMakeFiles. Line 10 creates a folder called build, and with line 11 the active folder changes to the BUILD folder. Line 12 executes CMake with the CMakeList in the ROOT folder. Now all makefiles are written to the BUILD folder. With line 13 make is executed and the GORILLA library and the executable are built in the BUILD/SRC folder.

```
1  #!/bin/bash -f
2
3  if [[ -d BUILD ]]
4  then
5      rm -rf BUILD
6  fi
7
8  export GORILLA_COVERAGE=FALSE
9
10 mkdir -p BUILD
11 cd BUILD
12 cmake ..
13 make -j
```

Listing 4: ROOT/build.sh

5.2 Implementation of pFUnit

The first step to implement pFUnit is to download and install pFUnit according to the two sections “Obtaining pFUnit” and “Building and installing pFUnit” on the GitHub page[22]. Afterwards, the user has to set the *PFUNIT_DIR* environment variable. This variable contains the path to the installation folder of pFUnit, which is (if not overwritten during the installation process) <pFUnit ROOT>/build/installed.
The next step is to enable pFUnit in the CMakeLists.txt in the ROOT folder of GORILLA. This is shown in line 17 to 20 in listing 2. The first line checks, if the *GORILLA_COVERAGE* environment variable is set to TRUE. The *GORILLA_COVERAGE* variable is set in the build_coverage.sh bash script (see listing 8). Afterwards, CMake has to search for pFUnit with the *find_package* command. The last step is to enable testing with the *enable_testing* command[43]. In line 62 to 64 of the CMakeLists.txt in the SRC folder (see listing 3) the folder to the test source file is added. Additionally, the GORILLA library gets the −−*coverage*

compiler option set in line 45 to 49, which is a synonym for *-fprofile-arcs -ftest-coverage* during compiling and *-lgcov* during linking. To see more details on the compiler options, take a look at the official manual of the GNU GCC compiler[44]. In the test folder there are a CMakeLists.txt (see listing 5) and two source files of the tests which were implemented until now. In the CmakeLists.txt is the declaration of the tests. To add a test, the command *add_pfunit_ctest* is used. In the scope of the command the name of the module, the source file for the test and the library to which it has to be linked has to be set. In this case, there are two test source files which test two source files in the SRC folder.

```
add_pfunit_ctest (test_various_functions_mod
  TEST_SOURCES test_various_functions_mod.f90
  LINK_LIBRARIES GORILLA
  )

add_pfunit_ctest (test_binsrc
  TEST_SOURCES test_binsrc.f90
  LINK_LIBRARIES GORILLA
  )
```

Listing 5: ROOT/SRC/TESTS/CMakeLists.txt

5.3 Implemented unit tests

Test inverse matrix (dmatinv3)

The test_various_functions_mod.f90 in the TESTS folder is for testing the source file various_functions_mod.f90 in the SRC folder. In the source file is the declaration of a module called various_functions_mod, which includes a subroutine called *dmatinv3*. The subroutine takes a real 3x3 matrix and calculates the inverse of it. If this is not possible, it sets the *ierr* variable to 1 and produces an output, which tells the user that the matrix is singular. The test_various_functions_mod.f90 is shown in listing 6. The first few lines are to initialize the module name and which modules to use in this file, including funit, which is part of pFUnit. To declare a test with funit the *@test* command is used (see line 8 and line 29). In the contains scope are two different tests, one for testing if the process of calculating the inverse of a matrix is working (subroutine *test_dmatinv3*), and one which intentionally fails because *dmatinv3* is called with a singular matrix to invert (subroutine *test_dmatinv3_fail*).
In the first subroutine, four matrices are initialized. The matrix A is the identity matrix. The inverse of the identity matrix is the matrix itself. To test this, the subroutine *dmatinv3* is called with the matrix A as input and matrix B as output. In line 18 the resulting inverse matrix B is compared with the matrix A with the *@assertEqual* command of pFUnit. If the two matrices are not equal, the user receives an error message and the values of the first values, which are not equal, and their location in the matrices.
The matrix C has the following structure:

$$C = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

The inverse matrix D can be calculated by hand and is equal to:

$$D = \frac{1}{2}\begin{pmatrix} -1 & 1 & 1 \\ 1 & -1 & 1 \\ 1 & 1 & -1 \end{pmatrix}$$

dmatinv3 is called with the matrix C as input and the result is saved in D. To check if the result of the function is correct, the inverse matrix which was calculated by hand is compared with the matrix D. Again, the *@assertEqual* command is used to compare the matrices.
In the test subroutine *test_dmatinv3_fails* the following matrix is used as input for *dmatinv3*:

$$A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

This matrix has no inverse matrix, because the last row only consists of zeroes. Then the *dmatinv3* subroutine is called with A as input. Now the subroutine should fail, and the *ierr* variable is set to 1. To check this, the *@assertEqual* command is used, in which the *ierr* variable is compared to 1.

```
module test_various_functions_mod
   use various_functions_mod
   use funit
   implicit none

contains

   @test
   subroutine test_dmatinv3()

      double precision,dimension(3,3) :: A, B, C, D
      double precision,dimension(3,3) :: D_known
      integer :: ierr

      A = reshape((/1.d0, 0.d0, 0.d0, 0.d0, 1.d0, 0.d0, 0.d0, 0.d0, 1.d0/),
    (/3, 3/))
      call dmatinv3(A, B, ierr)

      @assertEqual(A, B, tolerance=1e-13, message="test various_functions_mod
    with identity matrix")

      C = reshape((/0.d0, 1.d0, 1.d0, 1.d0, 0.d0, 1.d0, 1.d0, 1.d0, 0.d0/),
    (/3, 3/))
      call dmatinv3(C, D, ierr)

      D_known = 1.d0/2.d0*reshape((/-1.d0, 1.d0, 1.d0, 1.d0, -1.d0, 1.d0, 1.d0
    , 1.d0, -1.d0/), (/3, 3/))
      @assertEqual(D, D_known, tolerance=1e-13, message="test
    various_functions_mod with a symmetric matrix")

   end subroutine test_dmatinv3

   @test
   subroutine test_dmatinv3_fail()

```

```
      double precision,dimension(3,3) :: A, B, C, D
      integer :: ierr

      A = reshape((/1.d0, 0.d0, 0.d0, 0.d0, 1.d0, 0.d0, 0.d0, 0.d0, 0.d0/), (/3, 3/))
      call dmatinv3(A, B, ierr)

      @assertEqual(ierr, 1, message = "test various_functions_mod, matrix not invertible")


   end subroutine test_dmatinv3_fail

end module test_various_functions_mod
```

Listing 6: ROOT/SRC/TESTS/test_various_functions_mod.f90

Test binary search (binsrc)

The test_binsrc.f90 in the TESTS folder is for testing the binsrc.f90 file in the SRC folder. The binsrc.f90 file consists of a subroutine called *binsrc*. This subroutine takes an array p of increasing numbers with dimension n and a number xi as input, and calculates the index i which satisfies the condition $p(i-1) < xi < p(i)$. The source code of test_binsrc.f90 is shown in listing 7. It includes a test subroutine *test_binsrc_1*. In this subroutine, an array p is initialized with the following values:

$$p = \begin{pmatrix} 1.5 & 2.5 & 3.5 & 4.5 & 5.5 & 6.5 & 7.5 & 8.5 & 9.5 & 10.5 \end{pmatrix}$$

The value for xi is set to 5. So in this case, the return value for the index i should be 5. To test this, the subroutine binsrc is called with the array p, the lowest index, the highest index, the value xi and the return value i. The returned value then is compared to 5 with the *@assertEqual* command in line 17.

```
1  module test_binsrc
2     use funit
3     implicit none
4
5  contains
6
7     @test
8     subroutine test_binsrc_1()
9
10       double precision, dimension(10) :: p
11       double precision :: xi = 5.
12       integer :: i
13
14       p = [1.5d0, 2.5d0, 3.5d0, 4.5d0, 5.5d0, 6.5d0, 7.5d0, 8.5d0, 9.5d0, 10.5d0]
15       call binsrc(p, 1, 10, xi, i)
16
17       @assertEqual(5, i, message = "test binsrc")
18
19    end subroutine test_binsrc_1
20
21 end module test_binsrc
```

Listing 7: ROOT/SRC/TESTS/test_binsrc.f90

5.4 Code Coverage

Now that the CMake files and the test source files are prepared, GORILLA can be built with the build_coverage.sh script in the ROOT folder (see listing 8). This file is similar to the build.sh file shown in listing 4, but enables the tests via the *GORILLA_COVERAGE* variable and generates the code coverage report. In line 8 of this file, the *GORILLA_COVERAGE* environment variable is set to TRUE. Additionally, CMake gets the information on where to find pFUnit in line 12 with the *-DCMAKE_PREFIX_PATH* option[45]. After executing the *make* command in line 13 everything is built. The module files, the GORILLA library and the executable can then be found in the BUILD/SRC folder. The object files are included in the GORILLA library and in the folder BUILD/SRC/CMakeFiles/GORILLA.dir. In the second mentioned folder are also files, which have the same names as the source files, but the file extension is *.gcno[46]. These files are created during the compilation process only if the *-ftest-coverage* option is used (or in this case the *−−coverage* option, which is a synonym for this and two other options[44]). After executing the tests with the *ctest* command in line 15 two additional files are created in the same folder with *.gcda file extension. The executables for the test are in the BUILD/SRC/TESTS folder and can also get executed manually. The *.gcda files hold the information, how often a line was executed in the two files which were tested. Together with the *.gcno files, these files are used to create *.gcov files with the *gcov-9* command in line 18. These *.gcov files are copies of the source code of the file, but with the information how often a line was executed. It is possible to read those files with a simple text editor. To improve readability, the files are transformed to a *.html web page. This is done with line 19 to 22 in the build_coverage.sh file. Line 19 processes the files, which were tested and line 20 processes the files which were not tested. The results are two files, covered.info and uncovered.info. With the command in line 21 these files are combined and the *genhtml* command generates the *.html files in the BUILD/COVERAGE folder.

```
1  #!/bin/bash -f
2
3  if [[ -d BUILD ]]
4  then
5      rm -rf BUILD
6  fi
7
8  export GORILLA_COVERAGE=TRUE
9
10 mkdir -p BUILD
11 cd BUILD
12 cmake .. -DCMAKE_PREFIX_PATH=$PFUNIT_DIR
13 make -j
14
15 ctest --output-on-failure
16 cd SRC/CMakeFiles/GORILLA.dir/
17
18 gcov-9 *.gcno
19 lcov --gcov-tool gcov-9 --capture --no-recursion --directory . --output-file
       covered.info
20 lcov --gcov-tool gcov-9 --capture --no-recursion -i --directory . --output-
       file uncovered.info
21 lcov -a covered.info -a uncovered.info --output-file result.info
22 genhtml --output-directory ../../../COVERAGE result.info
```

Listing 8: ROOT/build_coverage.sh

Examples of the resulting *.html files are displayed in figure 2 and 3. Figure 2 is an overview of the code coverage of all source files. Figure 3 shows how often a line of ROOT/SRC/binsrc.f90 was executed.

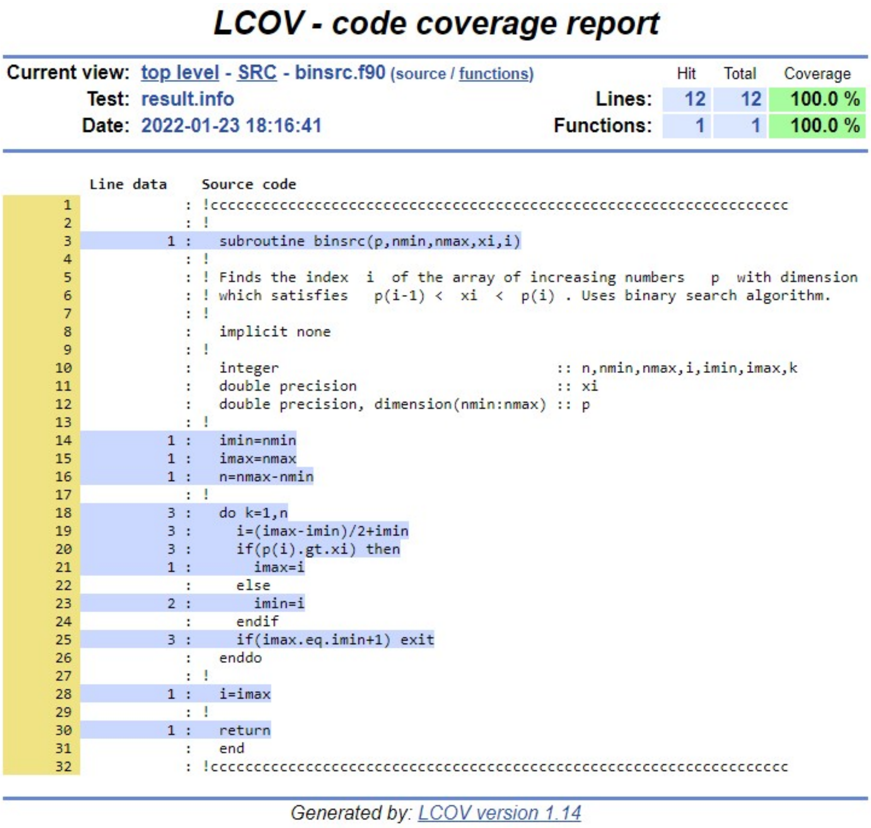
LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level - SRC				
Test:	result.info	**Lines:**	31	7026	0.4 %
Date:	2022-01-23 18:16:41	**Functions:**	2	230	0.9 %

Filename	Line Coverage			Functions	
bdivfree.f90		0.0 %	0 / 950	0.0 %	0 / 17
binsrc.f90		100.0 %	12 / 12	100.0 %	1 / 1
chamb_m.f90		0.0 %	0 / 5	0.0 %	0 / 1
circular_mesh.f90		0.0 %	0 / 274	0.0 %	0 / 12
differentiate.f90		0.0 %	0 / 26	0.0 %	0 / 1
field_divB0.f90		0.0 %	0 / 627	0.0 %	0 / 20
field_line_integration_for_SYNCH.f90		0.0 %	0 / 165	0.0 %	0 / 4
get_canonical_coordinates.f90		0.0 %	0 / 297	0.0 %	0 / 6
gorilla_plot_mod.f90		0.0 %	0 / 226	0.0 %	0 / 9
gorilla_settings_mod.f90		0.0 %	0 / 9	0.0 %	0 / 2
magdata_in_symfluxcoord.f90		0.0 %	0 / 109	0.0 %	0 / 2
magfie.f90		0.0 %	0 / 102	0.0 %	0 / 2
make_grid_rect.f90		0.0 %	0 / 231	0.0 %	0 / 2
nctools_module.f90		0.0 %	0 / 46	0.0 %	0 / 11
new_vmec_allocation_stuff.f90		0.0 %	0 / 16	0.0 %	0 / 2
odeint_rkf45.f90		0.0 %	0 / 17	0.0 %	0 / 1
orbit_timestep_gorilla.f90		0.0 %	0 / 135	0.0 %	0 / 9
plag_coeff.f90		0.0 %	0 / 23	0.0 %	0 / 1
points_2d.f90		0.0 %	0 / 125	0.0 %	0 / 3
preload_for_SYNCH.f90		0.0 %	0 / 40	0.0 %	0 / 1
pusher_tetra_func_mod.f90		0.0 %	0 / 37	0.0 %	0 / 1
pusher_tetra_poly.f90		0.0 %	0 / 912	0.0 %	0 / 30
pusher_tetra_rk.f90		0.0 %	0 / 1209	0.0 %	0 / 32
runge_kutta_mod.f90		0.0 %	0 / 24	0.0 %	0 / 2
scaling_r_theta.f90		0.0 %	0 / 8	0.0 %	0 / 2
spl_three_to_five_mod.f90		0.0 %	0 / 449	0.0 %	0 / 9
spline5_RZ.f90		0.0 %	0 / 185	0.0 %	0 / 3
spline_vmec_data.f90		0.0 %	0 / 264	0.0 %	0 / 7
supporting_functions_mod.f90		0.0 %	0 / 95	0.0 %	0 / 6
tetra_grid_mod.f90		0.0 %	0 / 105	0.0 %	0 / 5
tetra_grid_settings_mod.f90		0.0 %	0 / 12	0.0 %	0 / 3
tetra_physics_mod.f90		0.0 %	0 / 171	0.0 %	0 / 11
tetra_physics_poly_precomp_mod.f90		0.0 %	0 / 71	0.0 %	0 / 9
various_functions_mod.f90		100.0 %	19 / 19	100.0 %	1 / 1
vmecinm_m.f90		0.0 %	0 / 30	0.0 %	0 / 2

Generated by: LCOV version 1.14

Figure 2: Code coverage overview over the source files


LCOV - code coverage report

Current view:	top level - SRC - binsrc.f90 (source / functions)		Hit	Total	Coverage
Test:	result.info	Lines:	12	12	100.0 %
Date:	2022-01-23 18:16:41	Functions:	1	1	100.0 %

```
Line data    Source code
 1        : !ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
 2        : !
 3      1 :   subroutine binsrc(p,nmin,nmax,xi,i)
 4        : !
 5        : ! Finds the index  i  of the array of increasing numbers   p  with dimension
 6        : ! which satisfies   p(i-1) <  xi  <  p(i) . Uses binary search algorithm.
 7        : !
 8        :   implicit none
 9        : !
10        :   integer                                :: n,nmin,nmax,i,imin,imax,k
11        :   double precision                       :: xi
12        :   double precision, dimension(nmin:nmax) :: p
13        : !
14      1 :   imin=nmin
15      1 :   imax=nmax
16      1 :   n=nmax-nmin
17        : !
18      3 :   do k=1,n
19      3 :     i=(imax-imin)/2+imin
20      3 :     if(p(i).gt.xi) then
21      1 :       imax=i
22        :     else
23      2 :       imin=i
24        :     endif
25      3 :     if(imax.eq.imin+1) exit
26        :   enddo
27        : !
28      1 :   i=imax
29        : !
30      1 :   return
31        :   end
32        : !ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
```

Generated by: LCOV version 1.14

Figure 3: Code coverage of ROOT/SRC/binsrc.f90

5.5 GitHub Workflows

Two additional GitHub workflows called Ubuntu.yml and Mac.yml were implemented, which are in the folder ROOT/.github/workflows. Both are similar except of the *PFUNIT_DIR* environment variable and the implementation of the dependencies. The workflows are described with the Ubuntu.yml file, shown in listing 9 below. The first line is only the name of the workflow. Line 3 to 9 is the information for GitHub, when to start the workflow. In this case the workflow is started when pushing (line 4 and 5), on pull_request (line 6 and 7) and the workflow can get executed manually (line 9). Two jobs are implemented, one is called "Ubuntu-coverage" (starts at line 12) and one is called "Ubuntu" (starts at line 57). The first part is the same in both jobs, telling GitHub which operating system it has to use. In the "env" scope, the environment variables are set. The step with the name "Checkout" is for accessing the repository. Afterwards, the required libraries and programs are installed in the step "Dependencies" and "Install pFUnit" (this step is only in the Ubuntu-coverage job). The "Additional Files" step is to download and use external libraries, which is described in the README.md in the paragraph "Include external library". Then the Build is executed with the prepared bash scripts in the step "Build". The job "Ubuntu-coverage" ends with exporting the coverage data as artifact. In the "Ubuntu" job, the examples are executed in the "Run examples" and in the "Run script" step. To run the fifth example, Matlab is required. To set up Matlab, see step "Set up Matlab"and the official MATLAB Actions page on GitHub[47].

```
name: Ubuntu

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

  workflow_dispatch:

jobs:
  Ubuntu-coverage:
    runs-on: ubuntu-20.04

    env:
      FC: gfortran
      PFUNIT_DIR: /home/runner/work/GORILLA/GORILLA/pFUnit/build/installed/
PFUNIT-4.2

    steps:
      - name: Checkout
        uses: actions/checkout@v2

      - name: Dependencies
        run: |
          sudo apt-get update
          sudo apt-get install wget unzip gfortran liblapack-dev libnetcdff-
dev
          sudo apt install lcov

      - name: Install pFUnit
        run: |
          git clone https://github.com/Goddard-Fortran-Ecosystem/pFUnit
          cd pFUnit
          mkdir -p build
          cd build
          cmake ..
          make -j$(nproc)
          make tests
          make install

      - name: Additional Files
        run: |
          cd $GITHUB_WORKSPACE
          wget -O 954.zip "https://dl.acm.org/action/downloadSupplement?doi
=10.1145%2F2699468&file=954.zip&download=true"
          unzip 954.zip
          cp 954/F90/Src/Polynomial234RootSolvers.f90 SRC/contrib/

      - name: Build
        run: |
          ./build_coverage.x

      - name: Archive code coverage results
        uses: actions/upload-artifact@v2
        with:
          name: code-coverage-report
```

```
          path: BUILD/COVERAGE/

  Ubuntu:
    runs-on: ubuntu-20.04

    env:
      FC: gfortran

    steps:
      - name: Checkout
        uses: actions/checkout@v2

      - name: Dependencies
        run: |
          sudo apt-get update
          sudo apt-get install wget unzip gfortran liblapack-dev libnetcdff-dev

      - name: Additional Files
        run: |
          cd $GITHUB_WORKSPACE
          wget -O 954.zip "https://dl.acm.org/action/downloadSupplement?doi=10.1145%2F2699468&file=954.zip&download=true"
          unzip 954.zip
          cp 954/F90/Src/Polynomial234RootSolvers.f90 SRC/contrib/

      - name: Build
        run: |
          ./build.x

      - name: Run examples
        run: |
          cd EXAMPLES/example_1
          ./test_gorilla_main_cmake.x
          cd ../../EXAMPLES/example_2
          ./test_gorilla_main_cmake.x
          cd ../../EXAMPLES/example_3
          ./test_gorilla_main_cmake.x
          cd ../../EXAMPLES/example_4
          ./test_gorilla_main_cmake.x

      - name: Set up MATLAB
        uses: matlab-actions/setup-matlab@v1

      - name: Run script
        uses: matlab-actions/run-command@v1
        with:
          command: cd MATLAB, example_5_cmake
```

Listing 9: ROOT/.github/workflows/Ubuntu.yml

6 Conclusion and Outlook

Until now, only a small part of the source code is tested, but the framework to implement unit tests is set up. The next step is to add more unit tests to raise the percentage of tested code. To add additional unit tests, it is only necessary to write a new test in a Fortran file in the ROOT/SRC/TESTS folder and to add the test to the CMakeList in the same folder. The result of this additional test is automatically added to the code coverage report due to the capabilities of CMake and the build bash script. The capabilities of pFUnit still get extended due to a fairly large community and active developers, and a lot of features of it remain unused in the GORILLA project until now. Furthermore, there is a new unit test framework in process by the community of `https://fortran-lang.org/`. The unit test framework is called test-drive and can get obtained on their GitHub page[48] and is worth a look for the future.

List of Figures

Listings

References

[1] Michael Eder, Christopher G. Albert, Lukas M. P. Bauer, Sergei V. Kasilov, Winfried Kernbichler, Markus Meisterhofer, Michael Scheidt.
GORILLA - Guiding-center ORbit Integration with Local Linearization Approach. Website.
`https://github.com/itpplasma/GORILLA`
Retrieved 12 February 2022.

[2] Graz University of Technology.
Plasma physics group of Graz University of Technology. Website.
`https://www.tugraz.at/institute/itpcp/research/plasma-physics/`
Retrieved 12 February 2022.

[3] Christopher Albert, Sergei V. Kasilov, Winfried Kernbichler.
SIMPLE - Symplectic Integration Methods for Particle Loss Estimation. Website.
`https://github.com/itpplasma/SIMPLE`
Retrieved 12 February 2022.

[4] Daniel Forstenlechner.
Fortran-Unit-Test-Trivial. Website.
`https://github.com/Forsti5/Fortran-Unit-Test-Trivial`
Retrieved 12 February 2022.

[5] Daniel Forstenlechner.
Fortran-Unit-Test-Basic. Website.
`https://github.com/Forsti5/Fortran-Unit-Test-Basic`
Retrieved 12 February 2022.

[6] Michael Eder, Christopher G. Albert, Lukas M. P. Bauer, Sergei V. Kasilov, Winfried Kernbichler, Markus Meisterhofer, Michael Scheidt.
GORILLA: Guiding-center ORbit Integration with Local Linearization Approac. Website.
`https://github.com/itpplasma/GORILLA/blob/main/PAPER_JOSS/paper.md`
Retrieved 12 February 2022.

[7] M. Eder, C. G. Albert, L. M. P. Bauer, S. V. Kasilov, and W. Kernbichler. Quasi-geometric integration of guiding-center orbits in piecewise linear toroidal fields, 2020. Physics of Plasmas, 27(12), 122508. `https://doi.org/10.1063/5.0022117`.

[8] M. Eder, C. G. Albert, L. M. P. Bauer, S. V. Kasilov, W. Kernbichler, and M. Meisterhofer. Gorilla: Guiding-center orbit integration with local linearization approach., 2021. Zenodo. `https://doi.org/10.5281/zenodo.4593661`.

[9] Paul Hamill. *Unit Test Frameworks.* O'Reilly Media, Inc, 2004.

[10] Free Software Foundation, Inc. GNU Make. Website.
`https://www.gnu.org/software/make/`
Retrieved 17 January 2022.

[11] Dongsoo Lee. make. Website.
`http://linux-command.org/en/make.html`
Retrieved 19 February 2022.

[12] Canonical Ltd. Install make-system-user on Ubuntu. Website.
`https://snapcraft.io/install/make-system-user/ubuntu`
Retrieved 19 February 2022.

[13] Homebrew Formulae. make. Website.
`https://formulae.brew.sh/formula/make`
Retrieved 19 February 2022.

[14] Kitware, Inc. and Contributors. CMake. Website.
`https://cmake.org/`
Retrieved 17 January 2022.

[15] Kitware, Inc. and Contributors. CMake Tutorial. Website.
`https://cmake.org/cmake/help/latest/guide/tutorial/`
Retrieved 17 January 2022.

[16] Kitware, Inc. and Contributors. Testing With CMake and CTest. Website.
`https://cmake.org/cmake/help/book/mastering-cmake/chapter/Testing%20With%20CMake%20and%20CTest.html`
Retrieved 20 March 2022.

[17] Kitware, Inc. and Contributors. CMake ctest executable. Website.
`https://cmake.org/cmake/help/latest/manual/ctest.1.html`
Retrieved 17 January 2022.

[18] Kitware, Inc. and Contributors. CMake Source Code. Website.
`https://github.com/Kitware/CMake`
Retrieved 17 January 2022.

[19] Kitware, Inc. and Contributors. CMake Installation Guide. Website.
`https://cmake.org/install/#download-verification`
Retrieved 17 January 2022.

[20] Unit testing frameworks in Fortran. Website.
`https://fortranwiki.org/fortran/show/Unit+testing+frameworks`
Retrieved 12 February 2022.

[21] List of unit testing frameworks. Website.
`https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks#Fortran`
Retrieved 12 February 2022.

[22] NASA and NGC TASC. pFUnit Github. Website.
`https://github.com/Goddard-Fortran-Ecosystem/pFUnit`
Retrieved 17 January 2022.

[23] Free Software Foundation, Inc. Introduction to gcov. Website.
`https://gcc.gnu.org/onlinedocs/gcc/Gcov-Intro.html`
Retrieved 17 January 2022.

[24] Free Software Foundation, Inc. Installing GCC. Website.
`https://gcc.gnu.org/install/`
Retrieved 17 January 2022.

[25] Linux Test Project. Lcov Github. Website.
`https://github.com/linux-test-project/lcov`
Retrieved 17 January 2022.

[26] Free Software Foundation, Inc. GCC, the GNU Compiler Collection. Website.
`https://www.gnu.org/software/gcc/`
Retrieved 17 January 2022.

[27] Kitware, Inc. and Contributors. CMake cmake_minimum_required command. Website.
`https://cmake.org/cmake/help/latest/command/cmake_minimum_required.html`
Retrieved 17 January 2022.

[28] Kitware, Inc. and Contributors. CMake project command. Website.
`https://cmake.org/cmake/help/latest/command/project.html`
Retrieved 17 January 2022.

[29] Kitware, Inc. and Contributors. CMake include_directories command. Website.
`https://cmake.org/cmake/help/latest/command/include_directories.html`
Retrieved 17 January 2022.

[30] Kitware, Inc. and Contributors. CMake find_package command. Website.
`https://cmake.org/cmake/help/latest/command/find_package.html`
Retrieved 17 January 2022.

[31] Kitware, Inc. and Contributors. CMake add_compile_options command. Website.
`https://cmake.org/cmake/help/latest/command/add_compile_options.html`
Retrieved 17 January 2022.

[32] Kitware, Inc. and Contributors. CMake add_link_options command. Website.
`https://cmake.org/cmake/help/latest/command/add_link_options.html`
Retrieved 17 January 2022.

[33] Kitware, Inc. and Contributors. CMake target_compile_options command. Website.
`https://cmake.org/cmake/help/latest/command/target_compile_options.html`
Retrieved 17 January 2022.

[34] Kitware, Inc. and Contributors. CMake target_link_options command. Website.
`https://cmake.org/cmake/help/latest/command/target_link_options.html`
Retrieved 17 January 2022.

[35] Kitware, Inc. and Contributors. CMake add_subdirectory command. Website.
`https://cmake.org/cmake/help/latest/command/add_subdirectory.html`
Retrieved 17 January 2022.

[36] Kitware, Inc. and Contributors. CMake add_library command. Website.
`https://cmake.org/cmake/help/latest/command/add_library.html`
Retrieved 17 January 2022.

[37] Kitware, Inc. and Contributors. CMake add_executable command. Website.
`https://cmake.org/cmake/help/latest/command/add_executable.html`
Retrieved 17 January 2022.

[38] Kitware, Inc. and Contributors. CMake target_link_libraries command. Website.
`https://cmake.org/cmake/help/latest/command/target_link_libraries.html`
Retrieved 17 January 2022.

[39] Kitware, Inc. and Contributors. CMake set_target_properties command. Website.
`https://cmake.org/cmake/help/latest/command/set_target_properties.html`
Retrieved 17 January 2022.

[40] Kitware, Inc. and Contributors. CMake Fortran_MODULE_DIRECTORY variable. Website.
`https://cmake.org/cmake/help/latest/prop_tgt/Fortran_MODULE_DIRECTORY.html`
Retrieved 17 January 2022.

[41] Kitware, Inc. and Contributors. CMake target_include_directories command. Website.
`https://cmake.org/cmake/help/latest/command/target_include_directories.html`
Retrieved 17 January 2022.

[42] Linux Shell Scripting Wiki. Shebang. Website.
`https://bash.cyberciti.biz/guide/Shebang`
Retrieved 16 March 2022.

[43] Kitware, Inc. and Contributors. CMake enable_testing command. Website.
`https://cmake.org/cmake/help/latest/command/enable_testing.html`
Retrieved 23 January 2022.

[44] Free Software Foundation, Inc. GNU GCC Compiler Options. Website.
`https://gcc.gnu.org/onlinedocs/gcc-9.3.0/gcc/Instrumentation-Options.html`
Retrieved 23 January 2022.

[45] Kitware, Inc. and Contributors. CMake CMAKE_PREFIX_PATH option. Website.
`https://cmake.org/cmake/help/latest/variable/CMAKE_PREFIX_PATH.html`
Retrieved 23 January 2022.

[46] Free Software Foundation, Inc. Gcov Data files. Website.
`https://gcc.gnu.org/onlinedocs/gcc/Gcov-Data-Files.html`
Retrieved 23 January 2022.

[47] Johan Pereira, mw-hrastega, MathWorks®. Use MATLAB with GitHub Actions. Website.
`https://github.com/matlab-actions/overview`
Retrieved 24 January 2022.

[48] Sebastian Ehlert, Jeremie Vandenplas. Fortran Unit Test Framework test-drive. Website.
`https://github.com/fortran-lang/test-drive`
Retrieved 24 January 2022.