Electronic Communications of the EASST
Volume 83 Year 2025

deRSE24 - Selected Contributions of the 4th Conference for Research Software Engineering in Germany

Edited by: Jan Bernoth, Florian Goth, Anna-Lena Lamprecht and Jan Linxweiler

Deploying a C++ Software with (or without) Python Embedding and Extension

Ammar Nejati, Mikhail Svechnikov, Joachim Wuttke

DOI: 10.14279/eceasst.v83.2596

Electronic Communications of the EASST (https://eceasst.org).
Published by **Berlin Universities Publishing**
(https://www.berlin-universities-publishing.de/)

Deploying a C++ Software with (or without) Python Embedding and Extension

Ammar Nejati[1] and Mikhail Svechnikov[2] and Joachim Wuttke[3]

[1] a.nejati@fz-juelich.de
[2] m.svechnikov@fz-juelich.de
[3] j.wuttke@fz-juelich.de
https://computing.mlz-garching.de
Forschungszentrum Jülich GmbH,
Jülich Centre for Neutron Science at Heinz Maier-Leibnitz Zentrum Garching,
Lichtenbergstraße 1, 85748 Garching, Germany

Abstract: We discuss the manifold difficulties in cross-platform software deployment. We first consider a pure C++ project. Then we discuss the additional problems that arise when a C++ core has an embedded Python interpreter and is exposed to Python with bindings automatically generated by Swig. We explain how such a software can be deployed to Windows, Linux, and macOS, in form of source archives, binary installers, packages for package managers, or Python wheels. Our solutions are based on proven experience with the physics software BornAgain.

Keywords: deployment, installer, packaging, DevOps, CI/CD, continuous delivery, cross-platform, cross-language, C++, Python, CMake, Swig.

1 Introduction

Making a software available to a multitude of external users is a hard problem because the target systems are diverse and unpredictable. Not only do they have different processors and other hardware components, but more importantly they are equipped with several layers of wildly diverse software, including operating system kernel, system libraries, package managers and other tools, in versions from obsolete to experimental, installed at diffferent, often inconsistent locations, and accessed through environment variables like `PATH` that can be misconfigured in any way. Software publishers soon learn from help requests that no assumption about reasonable computer configuration holds without exception.

For developers of open-source software the easiest solution is to release just the code and build scripts, and leave it to the users to compile the software on their target systems. We will review building from source in Sect. 2. Many users, however, will find it painful or outside their competence, especially if the configuration script starts complaining about unmet dependencies. Dissemination of the software will suffer in consequence.

The next simple solution is to statically link everything into a single executable — including copies of libraries that are already present on the target system, which wastes download bandwidth and disk space, requires more memory at run time, and may take more time to start. Yet weighed against developer effort it may in many cases be an acceptable compromise. With this paper we want to help those who need to go further.

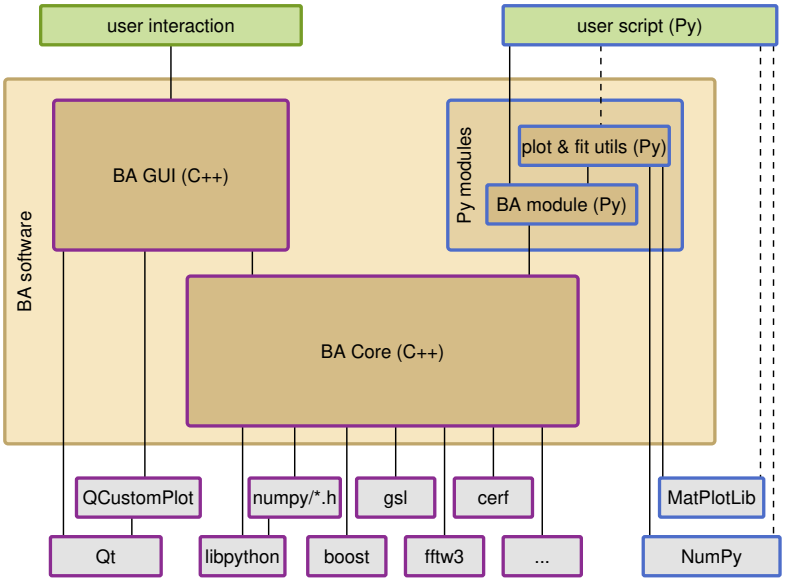

Figure 1: Architecture of the software BornAgain (BA) [1, 3]. The simulation and fit routines in Core are controlled through either the graphical interface or a Python script that imports the BA Python module. Optionally, the script may use plot and fit tools from extra modules in the BA Python package. Gray fields are external dependencies. Border colors indicate C/C++ versus Python components.

In Sect. 3, we discuss the next two levels of sophistication, namely binary installers and packages for given package managers. In Sect. 4, we add one extra requirement: the software shall be written in a compiled language, but also have a Python application programming interface (API). As we learned over the years, this requirement is at the origin of numerous complications.

This paper reflects experiences we made with the simulation software BornAgain[1] [1, 2, 3], which has a C++ kernel that can be run either from Python scripts or from a Qt-based graphical user interface (GUI) written in C++ (Fig. 1). Specifically, we will document the Python/C++ integration, the build procedure and the packaging for release 21.2 of May 2024. The source archive is available from the release page in the source repository [3], and has additionally been published with Zenodo [4].

2 Building Compiled Software from Source

For a software to be truly open, we must publish not only the source code in the narrow sense, but also all scripts that are needed to compile that code, and these scripts need to be configurable so that they do not depend on the specific setup of our own developer computers or build servers. In this section, we explain how we meet these requirements using the CMake/CTest/CPack tool

[1] BornAgain is an open-source research software for simulating and fitting neutron and x-ray reflectometry, off-specular scattering, and grazing-incident small-angle scattering. Its name alludes to a standard approximation in scattering theory, named after Max Born.

suite.

2.1 Build Automation with CMake

For all but the smallest software projects it is imperative to automatize the build process. We do not call the compiler and linker manually for all source files and binary objects but delegate this to the build engine Ninja, a lightweight replacement for the traditional Unix program Make that is also available for Windows and is faster than MSBuild. We combine Ninja with the tool Ccache (Linux, Mac) or BuildCache (Windows) that caches compilations and thereby saves much recompilation time in development.

The make script that steers Ninja (by default named `build.ninja`, the equivalent of the traditional `Makefile`) is written by a second-degree build automation tool, CMake.[2] CMake finds external dependencies (compiler, other tools, and libraries) and generates platform-specific build scripts. We also use CMake's companion programs CTest and CPack. Test automation with CTest is very important, but not directly related to deployment and shall therefore not be discussed here. CPack, on the other hand, is pertinent: we use it to build source and binary packages (for the latter, see Sect. 3.2.1).

To give a rough idea what fraction of the overall development effort goes into the build system: For the 85k lines of C++ and Python code in BornAgain, we have about 4k lines of CMake configuration scripts and 1k lines of installation related Python and shell scripts. These scripts are part of the source tree, available from the public repository or from source packages, and subject to the same license as the C++ and Python code.

2.2 Build Prerequisites

The build instructions for a complex software typically start with a list of required tools and libraries. The minimal tool suite consists of a C++ compiler, Ninja or equivalent (Make, NMake), and CMake. The list of external library dependencies is specific for each software project.

The simplest and recommended way of installation is by using a package manager that downloads packages from one or several standard repositories, like Debian's Dpkg or Redhat's Yum for Linux, Homebrew for macOS, and Vcpkg for Windows.[3] Typically, there are several packages for one library, like under Debian:

```
$ apt-cache search libcerf
libcerf-dev - Complex error function library - development files
libcerf-doc - Complex error function library - documentation
libcerf2 - Complex error function library - binary files
```

A Debian application package would just depend on package `libcerf2` that provides the shared library `libcerf.so.2.4`, whereas for compiling the same application from source

[2] CMake is a huge software with currently 870k lines of C++ source code. Rather than struggling with the online documentation (which is terse, with highly special terminology, yet does not cover all functionality in sufficient depth), it is a worthwhile investment to learn the basic concepts of CMake from a book. As the semi-official book *Mastering CMake* [5] is badly outdated, we rather recommend a self-published book by a CMake consultant and codeveloper [6].

[3] We have no practical experience with Vcpkg yet; in the past, we provided Windows packages of our own.

one needs package `libcerf-dev` that provides the provides the header (`.h`) files and entails package `libcerf2` as a dependency.

For the more exotic dependencies that are not available from all standard package repositories we include their source code in our source package, under the top-level directory called `3rdparty`. These dependencies are fully integrated in our CMake configuration, with build scripts either from upstream or added by ourselves.

2.3 Build Steps

The software to be built is typically obtained from the project's official download location as a source package, i.e. a compressed Tar archive (also called *tarball*, with file names like `BornAgain-21.2.tar.gz`) that contains a full copy of the source tree. Nowadays, however, colleagues who are motivated and confident enough to build a complex software from source could also be advised to clone the Git respository and check out the commit that represents the latest stable release. Therefore source packages are no longer strictly necessary, but may still be appreciated by some.[4]

After these preparations, the build process itself is very simple:[5]

```
cd <build_directory>
cmake ..
cmake --build .
ctest
cmake --install . # possibly under 'sudo'
```

If dependencies are missing, CMake will complain, and the process needs to be iterated.

While this is just standard for anybody with some practice in software development or system administration, it is frightening and difficult for many researchers with no such background. The installation of library dependencies is a huge obstacle especially under Windows where there is no well-established practice of using package managers. In consequence, the vast majority of our software users wants us to provide easy-to-use binary installers or packages, as described in the later sections of this paper.

2.4 Configuring by CMake

The make script (`build.ninja` or `Makefile`) that controls the compilation is not part of the project sources, but must be generated by CMake on the very same host computer on which the compilation shall take place. This is necessary because the make script contains numerous details that cannot be hard coded in the sources but depend on the software configuration of the host computer.

[4] Source archives can be generated through CMake's `package_sources` target. For projects that are hosted on a GitLab instance, tarballs of the entire source tree are automatically built and published for each release.

[5] Building in a dedicated subdirectory, typically called `build`, is strongly recommended. Our CMake scripts raise a fatal error if launched from the top-level source directory, as this is likely done by accident, and cumbersome to clean up.
Depending on `$PATH` and other system environment settings, the initial `cmake` command may need additional options, as can be seen in the file `.gitlab-ci.yml`, contained in the top-level BornAgain source directory, that steers our own continuous-integration builds.

By convention, files with name ending with `.in` are meant to be *configured* by CMake, as specified by a `configure_file` command in the project's CMake scripts. When `cmake` is run, then the file is copied to a new location with the trailing `.in` omitted from the file name, and with certain macros (typically written `@name@`) expanded to values set by CMake. In this way, we can inject for instance version numbers and directory paths into C++ sources or build scripts.

Another configuring performed by CMake is the search for external libraries and other dependencies. The search for a library is usually driven by a `find_package` command. The default variant of this command first searches at canonical locations for a CMake config file (like `cerfConfig.cmake`). If none is found then it searches for a *find modules* that is specific for the library or utility to be found. More than 150 find modules are shipped with CMake.[6] For other dependencies, including some fairly standard libraries like FFTW and TIFF, we must provide a find module of our own. Typically, we would not write such a CMake module from scratch, but reuse open-source code from the web (beware, however, that modern and clean CMake code is very rare). Additional search paths can be supplied to find modules via `cmake` options like `-DCMAKE_PREFIX_PATH`. This is particularly relevant for users who have no administrator privileges and therefore must install library dependencies under their own home directory.

3 Deploying C++ Binaries

To support deployment in binary form, software maintainers may provide installers or packages for different target platforms (Sect. 3.1). This is steered by CPack and further automatized by CI/CD (Sect. 3.2). Care must be taken to keep external file dependencies, especially shared libraries, findable (Sect. 3.3). Details differ for the three target operating systems Linux (Sect. 3.4), macOS (Sect. 3.5), and Windows 3.6).

3.1 Artifacts to be Provided

3.1.1 Installers versus Packages

An *installer* is a simple computer program that copies a software onto the target system. It may be parameterized through command-line options or through an interactive dialog that allows for instance to choose the install locations or to select modules to be installed. The installer is either self-extracting or triggers downloads from the web.

A *package* is a bundle of data or software, in a certain directory structure, with some metadata, for use with a specific package manager. Working with package managers is standard under Linux and frequent practice under macOS, but less common under Windows. Library dependencies are typically not packed in the application package, but in separate packages. The package metadata contain information about package dependencies, and thereby ensure consistency of all installed software.

Running a binary installer gives users the flexibility to install to arbitrary locations, either in a system directory (if they have write access) or in their home directory. The reverse side is the risk

[6] https://cmake.org/cmake/help/latest/manual/cmake-modules.7.html#find-modules.

that software installations become inconsistent and that directories become littered with unused or outdated binaries.

Installing software with a package manager typically requires administrator rights. This is problematic for a research software as it is not rare that a researcher urgently wants to try out a specific software at a time of day when no system administrator can be reached.

Typically, installers are provided on the download page of the project website, whereas it is more common for packages to be distributed through central repositories.

3.1.2 Target Platforms

Binary executables consist of machine instructions for a specific processor architecture. They also contain instructions that pass control to functions provided by libraries that are part of the operating system (like `libc.so` on Unix systems) or of the compiler (the C++ standard library `libstdc++.so`). Therefore, installers or packages must be specific for architecture and operating system. These days, in our field, it seems necessary and sufficient to support four combinations of these: x64 processors (also designated as x86_64 or AMD64) with Linux, macOS, or Windows, and ARM64-based Apple Silicon processors with macOS.

System libraries are designed with a strong emphasis on *backward compatibility*: Code that has been compiled against older versions will still work with newer versions of these libraries. The reverse is not true: Code compiled against recent library versions may fail on older target platforms. Therefore binary distributions must be built under the oldest operating system configuration one wants to support.

3.2 Creating Installers and Packages

3.2.1 CPack

To build binary installers or packages, one calls essentially

```
cmake .. [options]
cmake -build .
cpack -G <generator>
```

The CMake scripts, processed by the `cmake` command, contain `install` statements that mark certain build targets for later inclusion in binary installers or packages. These targets include the executable application, shared libraries, and other run-time file dependencies. In the `ninja` (or `make`) step, the targets are actually produced (compiled from source, configured, or just copied). In the `cpack` step, the selected files are packed into the installer specified by the CPack backend `<generator>`. Most generators are specific for one target operating system; therefore we discuss them further in Sects. 3.4 to 3.6.

3.2.2 Continuous Integration and Delivery

Modern DevOps practices [7, Sect. 3.2, and references therein], starting with *continuous integration* (CI), help to ensure the integrity of a software. Before any change is accepted into the main branch, the software must build on all target platforms, and all tests must pass. If this integration

test also comprises the creation of binary installers or packages, then a first step is made towards *continuous delivery* (CD).[7] For full CD, the binary artifacts ought to be unpacked and tested in a virtual environment.

In BornAgain, CI and CD are accomplished by GitLab runners. Details can be found in the control script `.gitlab-ci.yml` that is part of the public source tree.

3.3 File Dependencies

3.3.1 Internal versus External Dependencies

A research software usually consists not just of a single executable but depends on further files. Typical dependencies include shared libraries, plugins, or auxiliary programs; manual pages, examples and other documentation; initialization scripts and configuration files; databases.

File dependencies can be *internal* (deployed along with the main executable in the binary installer or package) or *external* (independently installed on the host system). Handling external dependencies is much easier with packages than with installers: If the external dependencies are available in separate packages, then the package manager ensures that consistent versions are installed at consistent locations. Conversely, binary installers are notoriously bad at handling external dependencies, and therefore should include whatever is needed to run the application — except for system libraries that are present on every host system, or for frameworks like Python that would cause difficulties if installed twice.

3.3.2 Install Locations

In the Unix tradition, there is a strong convention that installable artifacts go to specific subdirectories: `bin` for application executables, `lib` for shared libraries, `share` for data files. There is more variablity about the installation *prefix*, i.e. the absolute path the above subdirectories are attached to. Under Linux, package managers typically install to `/usr`, while locally compiled software goes to `/usr/local`. Centrally administered networks may have additional installation prefixes.

Alternatively, one may install outside the standard search paths. The prefix can then be chosen to contain name and version of the software, say `/home/me/bornagain-21`. This has several advantages: It simplifies removal. It allows several software versions to be installed on the same machine with no risk of interferences. Similarly, it prevents conflicts of library dependencies with other installed versions. And it does not require administrator privileges. The one disadvantage is that users need to add the `bin` and `lib` directories to the respective search paths for exectuables and shared libraries.

3.3.3 Shared Libraries

The problem of keeping file dependencies findable [8, ch. 7] is hardest for shared libraries because they cannot be configured in code or at run time but must be found when an application is launched. The only exception is when a shared library is loaded as a *plugin*, at runtime and

[7] Note that the abbreviation CD is also used for the related, but more ambitious concept of *continuous deployment* [7, Sect. 3.2.3, and references therein].

under the control of the user. However, plugins only make sense for application-specific add-on functionality. More basic libraries should be loaded at program startup.

Before we say more about the problem of installing shared libraries to findable locations, we shall provide some background on dynamic linking. To start, we must differentiate between two software tools associated with shared libraries: the *linker* and the *loader* [9].

The linker's main task is to bind symbolic names to memory addresses. In the case of a shared library, these are preliminary, relative addresses, starting from zero for each library. As the shared library may be used by different applications that may involve many other shared libraries, there is no way to prevent address conflicts at build time, and therefore the linker cannot assign definitive, absolute addresses [10, 11].

Execution of an application always begins in the kernel. The kernel reads some information from the header of the application binary, then passes control to the loader. If the application requires shared libraries, then the loader searches them, reads headers and symbol tables, and assigns absolute addresses for all symbols. This process, called *relocation*, is trivial for symbols that are defined in the same shared library where they are used (just add an offset to the relative address), but costly for external symbols, and therefore can cause noticeably delay upon starting a large application, which may warrant some optimization [11].

With this, we are ready to address the question how the loader searches for shared libraries. The answer depends on the operating system, and in particular on the format of executable binaries. Therefore we discuss it separately for each of our three target operating systems in Sects. 3.4 (Linux), 3.5 (macOS) and 3.6 (Windows), along with the generation of binary installers and packages.

3.4 Linux

3.4.1 Linker and Loader, Binary Format, Library Search

In the Unix world, the linker (`ld`) is sometimes called the *link editor*, and the loader (`ld.so`) is often termed the *dynamic linker*. Here we stay with the simple terms *linker* and *loader*. Executable binaries have the Executable Link Format (ELF) that was introduced in System V in the early 1990s.[8] An ELF file specifies library dependencies in the form

`[`*path*`/]lib`*name*`.so[.`*major*`[.`*minor*`[.`*patchlevel*`]]].`

The relative or absolute *path* is provided in special cases only. Normally it is left to the loader to search under certain paths for libraries that have the specified name and a compatible version number (same *major*, and same or later *minor*.*pathlevel*).

An ELF header may contain attributes `DT_RPATH` and `DT_RUNPATH` that specify search paths for library dependencies.[9] They have different rank in the overall search order. As a further difference, `DT_RPATH` propagates to dependent libraries, `DT_RUNPATH` does not. Therefore, for

[8] `man elf(5)`, and references therein. For the following, see also the man pages `ld.so(8)` and `ld(1)`.

[9] `DT_RUNPATH` was introduced later, and was meant to ultimately supplant `DT_RPATH`. In the years 2005-2024, the man pages `elf(5)` and `ld.so(8)` described `DT_RPATH` as deprecated, although there was no credible path towards its replacement and removal. While preparing this paper, we convinced the Linux man pages maintainers to undeprecate `DT_RPATH`.

a binary installer that includes indirectly dependent libraries, we need `DT_RPATH` rather than `DT_RUNPATH`.

Ignoring the latter, the loader will search for libraries in the following order:

1. in the directories specified in the `DT_RPATH` attribute of the ELF file (set through the *rpath* option of `ld` as explained below);
2. in the directories given by the environment variable `LD_LIBRARY_PATH`;
3. in the files listed in the binary resource `/etc/ld.so.cache` (which is a cache of `/etc/ld.so.conf`, created and updated by command `ldconfig`);
4. in the directories `/lib` and `/usr/lib`.

Directory `/lib` is reserved for system libraries. Installation to `/usr/lib` is not possible without root privileges, and may cause version conflicts with other installed software. Tweaking the search through `LD_LIBRARY_PATH` is a valid hack in development, but no good choice for deployment because users would need to give up control of their system configuration, and may run into inconsistencies, e.g. when working with symbolic links. Requesting users to change their `/etc/ld.so.conf` would be even worse. Therefore, the best (or least bad) solution is to specify the search path through `DT_RPATH`.

The ELF attribute `DT_RPATH` is set by the linker `ld` to hold directories specified by the options `-R` or `-rpath`. The paths given as arguments to these options may contain the special token `$ORIGIN` that is transmitted by the linker and expanded by the loader, which replaces it by the path where it found the depending binary. So we can install the application and its library dependencies in subdirectories `bin` and `lib` under whatever prefix, and make sure the libraries will be found by setting the rpath to `$ORIGIN/../lib`. Attributes in an ELF file can be inspected with the command `objdump -x` or `readelf -d`.

As the linker is under CMake control, the rpath is set by statements like[10]

```
target_link_options(target,
    PRIVATE "-Wl,--disable-new-dtags,-rpath=$ORIGIN/../lib")
```

The string constant is forwarded to the compiler. Flag `-Wl` instructs the compiler to forward the remainder of the string to the linker. Option `--disable-new-dtags` lets the linker set `DT_RPATH` instead of the newer `DT_RUNPATH`.

3.4.2 Installer

To build BornAgain for binary distribution, we use a docker container with Debian's *oldstable* release, which is lagging by two years behind *stable*. For some dependencies, we install newer versions from *stable-backports*, or build from source. Thereby we obtain binaries that work with Libc as old as version 2.31, released in 2/2020. This ensures forward compatibility even with the most conservative Linux installation encountered in any research institute we are cooperating with.

[10] Which, however, did not work under CMake $\leq$ 3.25 because of a bug in the handling of the dollar character.

The specific workflow for generating the binary installer starts after linking and testing locally. We run a shell script to postprocess our application binary and its dependencies. The script recursively walks through the directed acyclic graph of dependencies, following the rules of `ld.so`, using `ldd` to retrieve further dependencies.[11] Using simple hard-coded heuristics, each node in the graph is classified as a system, Python, Qt or other (regular) library dependency. System and Python dependencies are installation prerequisites for BornAgain and shall therefore not be included in our installer; at such nodes, the recursion does not proceed further. The application binary and all its regular and Qt library dependencies are copied to subdirectories of the package root directory.

From there they are taken by a CPack generator backend. For BornAgain, we have chosen the STGZ generator. It creates a self-extracting gzipped tar archive, i.e. a shell script that unpacks itself into a target directory with subdirectories `bin`, `lib`, `share`, as discussed in Sect. 3.3. This is the installer we publish on our Linux download site.[12]

3.4.3 Debian Package

Debian (`.deb`) is the most widely used package format for Linux. If resources are limited then it is a good choice to support just this one. Users of distributions based on other package managers will find ways to install Debian packages, for instance using the package converter Alien.[13] The BornAgain Debian package is kindly provided by external colleagues who have created a huge collection of software packages for our scientific field.[14]

These packages are of particular value for system administrators at large research facilities who have to deploy many domain-specific application to many computers. Debian offers them easy installation and high consistency. Individual users still might prefer compilation from source or binary installers to get more recent software versions than those in Debian/stable.

3.5 macOS

3.5.1 Linker and Loader, Binary Format, Library Search

In the Mac world, linker and loader are called *static linker* (`ld` or `libtool`) and *dynamic linker* (`dyld`). The native format for exectuable binaries is called Mach-O (object file format for the Mach kernel). Mach-O supports *shared libraries* (extension `.dylib` or `.so`) and *dynamic load modules* (also named *bundles*, extension `.bundle` or `.so`). The latter are mostly meant for plugins, and shall not be considered here.

The linker `ld` writes for each shared library dependency a `LC_LOAD_DYLIB` attribute to the Mach-O output file. The attribute's value consists of a *load command* with an argument that is

[11] While preparing this paper, we discovered the CMake command `file(GET_RUNTIME_DEPENDENCIES ...)` that automatizes this retrieval.

[12] https://bornagainproject.org/ext/files/latest/linux_x64.

[13] https://sourceforge.net/projects/alien-pkg-convert.

[14] https://salsa.debian.org/pan-team/soleil-packaging-overview, the photon and neutron Debian team at synchrotron Soleil. Though CPack has a Debian generator, they prefer their own packaging scripts, starting from a standard binary installation. They also push packages to the central Debian repository, https://packages.debian.org/stable/science/bornagain.

the *install name* (also called *including id*, *identification name*, or *install path*) of the dependent library. At variance from ELF, a Mach-O install name usually includes a path. Paths are either absolute (starting with `/`) or relative to one of the following prefix tokens:

- `@executable_path`: directory of the main executable of the current process;
- `@loader_path`: directory of the current binary (the one containing the load command), similar to the `$ORIGIN` token of ELF;
- `@rpath`: substituted with each path in a search list until a dylib is found.

The search list for `@rpath` is constructed from the paths stored in `LC_RPATH` attributes[15] of the dependency chain leading to the current library. As an additional complication, paths in `LC_RPATH` may contain `@executable_path` or `@loader_path` prefixes (but not `@rpath` as this would make the search circular). In BornAgain, we set `LC_RPATH` by the CMake command `target_link_options` (introduced in Sect. 3.4.1) that feeds option strings like

```
"-Wl,-rpath=@loader_path/../lib"
```

to the linker.

The linker takes the install name of a dependent library from its `LC_ID_DYLIB` attribute. Only if a library has no such attribute, its absolute path is taken. The `LC_ID_DYLIB` attribute is meant to contain the canonical installation path, and typically has an `@rpath` prefix (e.g., `@rpath/libfoo.dylib`). Therefore, its discovery by the dynamical loader involves the search list that is constructed from the `LC_RPATH` attributes of the libraries on the dependency graph.

3.5.2 Installer

As for Linux (Sect. 3.4.2), the generation of the binary installer starts after linking and testing locally. As the dependency postprocessing for macOS is quite involved, we steer it by a lengthy Python script. The script recursively walks through the graph of dependencies, following the rules of `dyld`, using `otool` to retrieve further dependencies. As for Linux, each node in the graph is classified as a system, Python, Qt or other (regular) library dependency. No action is taken for system and Python dependencies. The application binary and all its regular and Qt library dependencies are copied to subdirectories of the package root directory. In each of them, all references to dependencies (stored in `LC_LOAD_DYLIB` attributes) are replaced by relative paths under the `@rpath` prefix, using `install_name_tool`.

When this postprocessing is done, CPack is called. With the DragNDrop generator, it creates an Apple Disk Image with extension `.dmg`, which is a compressed copy of a directory tree. Historically, it was straightforward to install such a `.dmg` file on a Mac, either using the Finder or from the command line. However, newer macOS versions make it increasingly difficult to install third-party software that is not signed and notarized. To sign, developers need to buy a key from Apple. In the notarization step, Apple checks conformity with their security rules. Until macOS

[15] In close analogy with ELF (Sect. 3.4.1), Mach-O also supports `LC_RUNPATH` attributes, which we shall not consider here.

12, it was still possible to run unsigned software after confirming a warning message. For macOS 13, we currently know no workaround. We have no experience with signing and notarizing yet, and have not yet made up our minds whether we are ready to provide Mac binaries under these conditions.

3.5.3 Homebrew Package

We are currently investigating Homebrew formulae as an alternative to binary installers. The Homebrew package manager and repository[16] seems to be popular enough so that we can expect or request our users to have it installed on their Macs. It has the advantage over other package managers (Fink, MacPorts) that users do not need administrator rights. Mac users who want to build BornAgain from source are since long advised to use Homebrew for installing library dependencies (Sect. 2.2).

Originally, Homebrew only contained *formulae* that steer compilation on the target systems. Nowadays, the default mode of distribution is *bottles* that contain precompiled binaries. New formulae or bottles can be either submitted to Homebrew's central repository, the *core tap*, or made available for download elsewhere. Given the difficulties with relocating dependencies and signing and notarizing binaries, we tend to fall back to a formula so that users have to build our software on their own machines.

3.6 Windows

For historic reasons, we build BornAgain under Windows using the native Microsoft Visual Studio (MVS) toolchain. If we were to start anew, we might consider Mingw-w64 as a potentially simpler alternative. For sure, using MVS has the advantage that it spots some programming errors that are overlooked by the other two compilers we are using (GCC and Clang).

3.6.1 Linker and Loader, Binary Format, Library Search

The linker `link.exe` is part of MVS. Usually, it is invoked automatically as part of the compiler toolchain. At variance from Linux and macOS, the loader is no separate piece of software but is part of the Windows kernel.

The format for executable applications and DLLs is called Portable Executable (PE) [9]. It has no equivalent of `RPATH`. When an application is launched, DLLs are searched in the application directory, in system directories (`System32` and `SysWOW64`), and in the directories listed in the `PATH` environment variable. The search order depends on the "Safe DLL search mode" flag.[17] Registering DLLs in the Windows registry seems to be neither necessary nor helpful in our context.[18]

[16] https://brew.sh.

[17] https://learn.microsoft.com/en-us/windows/win32/dlls/dynamic-link-library-search-order.

[18] Registration is only requested when COM (Component Object Model) classes are involved (https://www.sevenforums.com/general-discussion/402100-registering-every-dll-required-possible.html#post3294192). COM is a Windows specific technology for inter-process communication that has no place in our cross-platform code.

3.6.2 Installer

While preparing this paper, we replaced the CPack generator for BornAgain. While the current version 21 still has an NSIS installer,[19] future releases will come with an installer generated by the Qt Installer Framework (IFW). While configuration of the NSIS installer is arcane,[20] configuration of the IFW generator is done with a handful of CMake variables and about 40 lines of JavaScript that install the desktop icon and a start-menu shortcut.

Both the NSIS and the Qt installer manipulate the Windows Registry. This is managed by CMake/CPack behind the scenes, and has caused no difficulties in the past.

3.6.3 No Managed Package

Installation from a package repository is no standard practice under Windows, nor is it clear which package manager may become more popular in the future: Chocolatey, or Microsoft's own Winget? Therefore we did not invest effort into preparing binary packages for Windows.

4 Deploying a C++ Software with Python API

The C++ core of our software BornAgain has access to the Python interpreter, *and* is partly exposed to Python (Fig. 1). Thereby we have proven experience with deploying software with both types of C++/Python interfaces.

4.1 C++/Python Interoperation

4.1.1 Embedding, Extending, Exposing: Terminology and Use Cases

A C program can be enabled to call Python functions and to manipulate Python data structures, if it has an *embedded* Python interpreter.

With an embedded interpreter, algorithms that are implemented only in Python become available from C++. This use case is increasingly frequent, as a growing body of research code is written in Python.

If functions and data structures from a C or C++ library are packed as a Python module, then this adds to the functionality available under Python, and therefore is described in the Python docs as *extending* Python. However, as developers of a heavy C++ core with a light Python layer we rather think of it as *exposing* our C++ code to Python.

A typical use case in a research context is a versatile simulation or data analysis framework that is written in a compiled language but can be steered through Python scripts. In this way, developers can take advantage of strong typing and of optimizing compilers, while users do not need to learn the low-level language and still have the full flexibility of programmatical control.

[19] Nullsoft Scriptable Install System (NSIS), originally developed by the defunct company Nullsoft, is now an open-source project with minimal maintenance activity.

[20] It involves an obscure script of almost 1000 lines that is copied since over twenty years from one open-source project to the next (proven by an uncorrected typo "Uninstall sutff" in a comment, and by another comment that refers to a script "Written by KiCHiK" from 2003). We therefore fear that the NSIS generator is not sustainable in the long term.

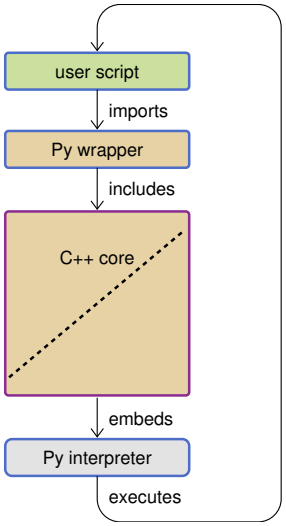

Figure 2: A sandwiched C++ core: It is exposed to Python through a wrapper, and has an embedded Python interpreter that runs user scripts that may import the Python API of the very same core. To prevent this from getting circular, the C++ functions that call the Python interpreter should *not* be exposed in the Python wrapper, as indicated by the dashed diagonal frontier.

Finally, BornAgain (Fig. 1) provides an example for a software that has both an embedded Python interpreter and a Python wrapper. This sandwich architecture must be used judiciously so that it does not become circular (Fig. 2). The embedded Python C API also allows to use native Python data structures, for instance NumPy arrays, in the C++ core, which facilitates the creation of a "pythonic" wrapper.

4.1.2 Embedding Python in C/C++

In our context it is safe to assume that the programming language Python is run through its reference implementation CPython, which is written in C and thereby provides a native C API. Thanks to the `extern` C linkage of C++, this API also works with C++. It is exposed in the header `Python.h`, which provides access to the C-based data structures and functions of CPython. To make the CPython interpreter callable from C or C++, the embedding code must include this header, and the compiled binary must be linked with the shared library `libpython`.[21] The header `numpy/arrayobject.h` gives access to NumPy data structures.[22]

Compared to a pure C++ program, an embedded Python interpreter poses no additional challenge to deployment because there is no interference between `libpython` and the standard Python interpreter on the host system.

[21] https://docs.python.org/3/extending/index.html and Ref. [12].

[22] This dependency will be broken by NumPy version 2. To deal with it once and for, in forthcoming BornAgain 22 all interaction with NumPy is moved from C to Python so that header `arrayobject.h` is no longer needed.

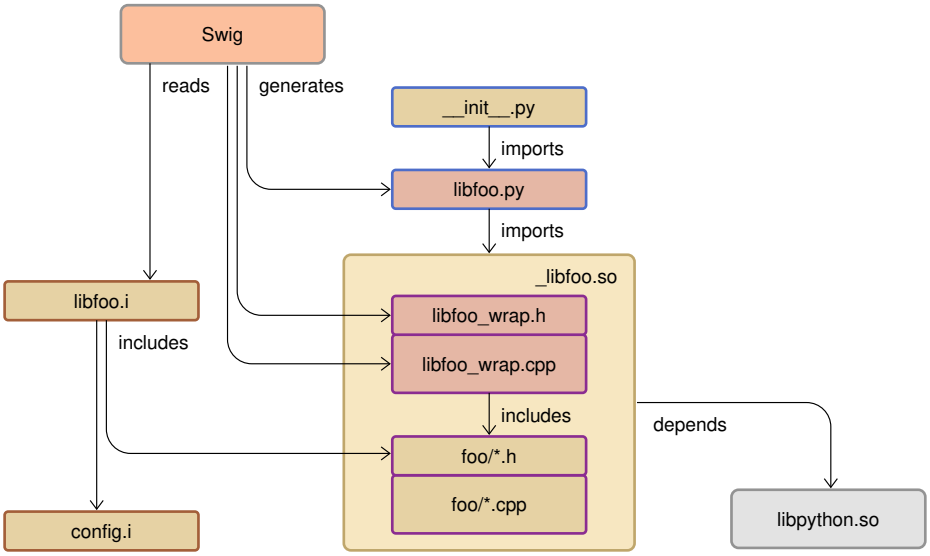

Figure 3: Generating Python bindings with Swig. Background colors differentiate user written code, automatically generated code, and external tools and components; border colors indicate C/C++ and Python sources as well as Swig configuration (.i) files. The C++ library Foo consists of sources `foo/*.cpp` and headers `foo/*.h`. Wrapper generation is steered by the Swig interface file `libfoo.i`.

4.1.3 Exposing C/C++ Code to Python

To expose a C or C++function to Python, one needs to write a wrapper function that converts function arguments and return values from Python to C/C++ and vice versa. Additionally, one needs to provide a module-methods table in C, and an initialization function in Python. Even for the simplest example, the instructions go over several pages [12].

For large projects it is out of question to write these wrappers manually. Indeed, several software solutions are available to automatize the wrapper generation. They attack the problem in different ways and at different software levels, they require different amounts of manual control code, they generate quite different amounts of boilerplate code in C/C++ or/and in Python, they differ in final execution speed, and they also differ in complexity, maturity, stability, and quality of documentation. The latter three factors weighed heavily in our decision to use Swig.[23]

Code generation with Swig is explained in Fig. 3. We consider a C++ library Foo that is made of sources `foo/*.cpp` and headers `foo/*.h`. Wrapper generation is steered by the Swig interface file `libfoo.i`, which containes a list of header files that shall be exposed to Python, plus some directives for handling templated classes and other special cases. The Swig executable

[23] https://www.swig.org. If we were to start anew, we would also consider *Shiboken*, solidly based on Clang and maintained by the Qt Group who use it to generate *Qt for Python*. Two other often named alternatives, *Pybind11* and *Cython*, certainly are mature but require an additional tool for the automatic generation of wrappers. Several projects address this need but it is not entirely clear if one of them has the quality and the momentum to cover all of C++ and to live on in the long run: we recently inspected and tested *Cppyy*, and were not convinced; maybe *Litgen*, though relatively novel, is a better candidate.

reads the interface file, and generates boilerplate wrapping code in C++ and in Python. All C++ sources, manually authored and automatically generated ones, must then be compiled and linked into a shared library `_libfoo.so`.[24] This library depends on `libpython.so`, as can be revealed with the Unix command `ldd`. It is ingested by `libfoo.py`, using Python's `import` statement. Finally, one may manually add `__init__.py` which consists of some statements like

```
from libfoo import *
```

to import the required functionality into the Python namespace.

The auto-generated `libfoo_wrap.cpp` can be huge, resulting in painfully long compilation times. Not least for this reason we split the BornAgain Core into several libraries. Each of them is wrapped separately as a Python module. All these modules are then wrapped as a single `bornagain` Python module.

Whether auto-generated data should be committed to version control is a difficult question that has no universal answer. In BornAgain, we opted for committing Swig-generated code to Git because it accelerates builds and reduces the number of software dependencies that need to be installed on our CI machines as well as on the machines of external users who want to compile BornAgain from source. The downside is that all developers are required to run the same version of Swig, lest the Git history be polluted by forth and back changes in the auto-generated code.[25]

4.2 Python Versions

4.2.1 Python Versions and ABI Incompatibility

As official support for Python 2 ended in 2020, we only consider major version 3.[26] However, we must be aware that binary-interface compatibility can be broken by minor releases.[27] Let us explain what that means, and more generally, what policies Python has on backward compatibility [14, 15].

Essentially, a new minor release will not break a working Python program, except if the previous release printed a warning that a deprecated language feature was used in that program [15]. As such deprecations are rare, and only concern very special language constructs, we can take for granted that the automatically generated Python API of our research software will continue to work throughout the life of Python 3.

The same holds for Python's C API. This means that a C/C++ program that includes the header `Python.h` will continue to compile if that header is replaced by a new version from a new minor release. However, *the resulting binary may change*, for instance because the memory layout of a data structure defined in `Python.h` has changed. The Python core developers have learned some lessons and are now taking care to minimize the frequency of such changes [14] so

[24] Here and the following, we use the Linux extension `.so` to designate shared libaries. For macOS or Windows, read `.dylib` or `.dll`, respectively.

[25] It is not even sufficient to work with one and the same point release of Swig: we found that Swig from Debian generates different (though functionally equivalent) code than Swig built from source.

[26] The forced migration from 2 to 3 caused severe discontentment and broke an unexpected number of software projects beyond repair [13, e.g.]. It seems that the lesson has been learned so that we need not to worry about major 4 any soon, https://answerpython.com/blog/why-python-4-may-never-arrive (2022).

[27] https://docs.python.org/3/c-api/stable.html.

that many minor releases do actually *not* break the application binary interface (ABI), but still we must be prepared that such changes may happen with any new minor.

The ABI comes into the game when a Python extension module `foo` is imported by a script or in an interactive session, and eventually the Python interpreter dynamically loads the shared library `_libfoo.so` as a plugin. As each extension library is linked with `libpython.so`, the latter is loaded as well. Following that, whenever function wrappers from `foo` are executed, the Python interpreter will call function binaries from `_libfoo.so`, and these may call to `libpython.so`.

Incompatibilities between the Python versions of the interpreter and `libpython.so`, or between `libpython.so` used at compile time and at run time, can result in segmentation faults, memory errors, or other low-level issues. Some of these issues may seemingly occur at random (heisenbugs), which makes them hard and painful to track down.

Further incompatibilities may arise from NumPy. NumPy has a C and a Python API, as indicated by the boxes labelled "numpy/*.h" and "NumPy" in Fig. 1. The ABI implied by the C API "is forward but not backward compatible. This means: binaries compiled against a given version of NumPy will still run correctly with newer NumPy versions, but not with older versions."[28]

4.2.2 Target Versions

Therefore different binary packages must be built, targetting all Python minor versions that are still in widespread use. For instance BornAgain 21 from summer 2023 provided binaries for Python 3.8 (first released in 2019) to 3.11.

This requirement is orthogonal to the ones from Sect. 3.1.2, namely that dedicated binaries must be provided for different processor architectures and operating systems. Altogether, this results in sixteen installers, with file names like

```
BornAgain-21.1-python3.11-mac_x64.dmg
```

for machines with an x64 processor, macOS and Python 3.11.

4.2.3 Switching between Python Versions

As developers, we need to create binaries for different Python minors, to test them, and to investigate issues reported by users. Therefore we need a convenient way to change the Python version that is active on our integration server and on our own workstation. Currently, our preferred solution is using the "simple Python version management" tool Pyenv[29] that is made for the sole purpose of managing isolated environments for different Python versions on the same computer.

4.3 Deploying a Python Package

In this section, we explain how to deploy a Python package that includes a binary library. It may suprise that we explored this way of software distribution for BornAgain, which has not

[28] https://numpy.org/doc/stable/dev/depending_on_numpy.html.

[29] https://github.com/pyenv/pyenv, not installable with Pip. Package *better-pyenv* on PyPI has a completely different purpose and should be ignored.

only a Python frontend, but also a GUI that is independent of Python (Fig. 1). However, some of our users are not interested in the GUI and requested us to provide a lightweight Python-only package (wheel) that can be downloaded from a central repository (PyPI, see Sect. 4.3.4), and are easy to install.

4.3.1 Python Wheel

A Python *wheel*[30] is a ZIP archive with regulated internal structure. It contains all the files that make up a Python package: metadata, Python code, and optionally compiled artifacts like the binary image of a shared library [16]. It is the recommended format for packages that contain binaries, and has advantages even for pure-Python packages [17].

A wheel has a name like

```
BornAgain-21.1-cp311-cp311-win_amd64.whl,
```

where `21.1` is the version of the packed software `BornAgain`, the first `cp311` is the target language version (CPython 3.11), the second `cp311` is the ABI version (see Sect. 4.2.1; `none` for a pure-Python package), and `win_amd64` indicates the target platform AMD64 with Windows operating system.

To inspect the contents of a wheel, one can use the command `unzip -l <wheel>`. One could also use `unzip` to install the wheel, but the preferred command is `pip install <wheel>`, provided by Pip, the package installer for Python. To find out where packages are going to be installed, use the Python interpreter to run

```
import sysconfig
print(sysconfig.get_path("purelib"))
print(sysconfig.get_path("scripts"))
```

The status of the package can be verified via `pip show <package-name>`, and the installation can be cleanly reverted with `pip uninstall <package-name>`.

4.3.2 Creating a Wheel

The package installer Pip is also the preferred tool for wheel creation, with the command

```
pip wheel <input-dir> --no-deps --wheel <output-dir>
```

Name and contents of the output wheel depend on parameters that are set through a control file `setup.py` or `pyproject.toml` that must be provided in the input directory. This files provides the interface between Pip and one of its backends (Distutils, Setuptools, Hatchling, Flit, PDM, Poetry).[31]

These backends were introduced at different times; they embody different design ideas, have different syntax, and support different hierarchies of differently named parameters. In BornAgain, we configure the former default backend Setuptools through the legacy configuration

[30] The backstory of the whimsical name "wheel" seems to be the following: In its early days, insiders referred to PyPI as "the cheese shop", after a Monty Python sketch. Now the shop is fully stocked with wheels of cheese.
[31] Section *Choosing a build backend* in [17].

files `setup.py` and `setup.cfg`. For a new project, it is recommended to use the modern control file `pyproject.toml`.[32]

4.3.3 The 'manylinux' Wheel

PyPI currently supports uploads of wheels only for the three target platforms Windows, macOS, and Manylinux [17]. The Manylinux specification [19] is a response to the difficulty of distributing pre-compiled binaries to various platforms that all run the Linux kernel but with different sets of user-space system libraries. To keep this challenge manageable, Manylinux only supports mainstream Linux distributions like Debian, OpenSuSE, Ubuntu, RHEL, etc., but not heavily patched platforms like Android.

Manylinux wheels only rely on a small set of system libraries that are available across all supported distributions. These are "glibc and a few others", for instance libncursesw [19]. The binaries in the wheel must be built against the oldest version of glibc that shall be supported, and this version number is part of the tag

```
manylinux_${GLIBCMAJOR}_${GLIBCMINOR}_${ARCH}
```

(e.g., `manylinux_2_35_x86_64`). All other shared library dependencies must be packed in the wheel, not unsimilar to what is done for Windows or macOS wheels [19].

While any method of producing Manylinux-compliant wheels is admissable [19], we follow the default recommendation to use Auditwheel,[33] a Python module that contains a set of tools and build images for building Manylinux wheels. Auditwheel inspects the binaries inside a "raw" wheel, and finds the dependencies on versioned external shared libraries. The command

```
auditwheel repair --plat <platform-tag> <wheel-name>
```

copies the required external shared libraries into the wheel to ensure, modifies the `RPATH` binary attributes, and adds a proper manylinux tag.

4.3.4 Publishing the Package on PyPI

PyPI, the Python Package Index, is the official repository for third-party Python packages.[34] It simplifies and streamlines the distribution, discovery, download and installation of Python packages, and gives cohesion to the community. It boasts a robust server infrastructure with reinforced security; project[35] maintainers need to register with two-factor authentication.

Each project has a landing page[36] that provides a project description, standardized metadata [21], and links to the wheel download page, to the release history, to the upstream project homepage, and more. Users can download and install packages from PyPI with a simple Pip command, like `pip install bornagain`.

[32] Defined in [18]. See also sections *pyproject.toml specification* and *Writing your pyproject.toml* in [17]. The Pip documentation v24.0, section Build System Interface, clearly says that `pyproject.toml` is prefered over the legacy `setup.py`.
[33] https://pypi.org/project/auditwheel.
[34] https://pypi.org, created in 2002 [20], now hosting over 500k projects.
[35] Terminology according to https://pypi.org/help: A *project* publishes *releases* that contain suites of *packages* for the different target platforms.
[36] E.g. https://pypi.org/project/BornAgain.

Projects should be maintained through personal user accounts. The maintainer starts the project, then associates other collaborators, just as in GitLab or GitHub. For each package, a private token is created by PyPI. This must be put in a configuration file which can then be used with the dedicated tool Twine[37] to upload a release to the PyPI repository.

5 Discussion

For a software to be adopted by a significant fraction of its potential usership, it must be installable with little effort. It is true that researchers are used to suffering, and ready to spend tremendous amounts of time with an inconvenient user interface if only the results are scientifically rewarding. But prior to this, they try out alternatives, if there are any, and the software that fails easy installation has lost within minutes.

An effortless user experience, however, requires considerable effort from the developers, as the present paper has amply shown. Part of the developer effort comes from the need to support different host operating systems. Cross-platform development itself has become relatively easy thanks to the thorough standardization of programming languages and the high quality of compilers, interpreters, and frameworks like Qt. This does not extend to deployment because of the deeply different philosophies of the target systems. As we have shown, there are good reasons why for different targets different artifacts should be provided. At best, some effort for multi-target deployment can be saved if the different installers or packages are generated out of cross-platform middleware like CMake/CPack or Qt Installer Framework. In BornAgain, we are using them, but not yet to the fullest possible extent.

Regarding the relative ease of deployment for different programming languages, Python wheels and their PyPI repository clearly beat whatever is available for C++. Deployment is well known to be a weak point of C++ and of compiled languages in general [22, starting at 33'50"]. Conversely, the weak point of Python is the fluidity of its "ecosystem": While the "Zen of Python" [23] plausibly requests that "there should be one — and preferably only one — obvious way to do it," there is a plethora of packaging levels and technologies, implemented by an ever changing landscape of specific software projects. Instead of *one* well designed solution being refined for some decades, every few years a new approach is hyped.[38]

Altogether, the tool stack presented in this work is of unequal quality: some solutions are elegant, solid and stable, while others feel overly complicated and brittle, like the low-level manipulation of rpaths. CMake and Swig are powerful tools, but of tremendous complexity; in spite of their extensive documentations, it is often hard to find out how they are meant to be used so that we had to ask in online forums and resort to trial and error.

Let us emphasize that what we have been presenting here is just one set of solutions to the cross-platform, cross-language deployment problem. Many other solutions are possible. That

[37] https://twine.readthedocs.io.

[38] E.g. https://packaging.python.org/en/latest/overview. The NumPy docs (https://numpy.org/install, section *Python Package Management*) argue that "managing packages is a challenging problem, and, as a result, there are lots of tools." Unfortunately, the converse is also true: Choosing the right tool from lots of similar ones, all with different shortcomings and unclear long-term perspectives, adds one more challenging problem to our stack. We failed at a first attempt to replace deprecated `setup.py` by `pyproject.toml`, and we are overwhelmed by the choice between the Pip backends Setuptools, Hatchling, Flit, PDM, Poetry.

we have said very little about possible alternatives in this paper is not a judgement about their possible merit but just a restraint not to talk about technologies we have no practical experience with. In favor of the solutions presented here, we only say that they are proven in practice, and can be freely inspected in an open-source repository. BornAgain is licensed under the GPL; if this is an impediment to the reuse of code, more liberal licenses can be negotiated.

With this paper, as with the actual engineering it describes, we had to adventure ourselves far beyond our comfort zone of consolidated knowledge and experience. It is not unlikely that this paper contains factual errors or describes unnecessary complications. We will be very grateful for any feedback. We intend to publish corrections and updates on the BornAgain website.

Acknowledgements: The solutions described in this paper have been worked out by a long sequence of BornAgain contributors.[39] We thank Emmanuel Farhi, Roland Mas and Frédéric-Emmanuel Picca for the BornAgain Debian package and for helpful communications. We are grateful to Brad King and Craig Scott for help with CMake.

References

[1] G. Pospelov, W. Van Herck, J. Burle, J. M. Carmona Loaiza, C. Durniak, J. M. Fisher, M. Ganeva, D. Yurov and J. Wuttke, *BornAgain: software for simulating and fitting grazing-incidence small-angle scattering*. J. Appl. Cryst. **53**, 262 (2020).

[2] A. Nejati, M. Svechnikov and J. Wuttke, *BornAgain, software for GISAS and reflectometry: Releases 1.17 to 20*. EPJ Web Conf. **286**, 06004 (2023).

[3] Forschungszentrum Jülich, Scientific Computing Group at MLZ, *BornAgain, open-source research software to simulate and fit neutron and x-ray reflectometry and grazing-incidence small-angle scattering.* Home page https://bornagainproject.org; repository https://jugit.fz-juelich.de/mlz/bornagain.

[4] A. Nejati, M. Puchner, M. Svechnikov and J. Wuttke, *BornAgain-v21.2 source archive, https://doi.org/10.5281/zenodo.13860826* (2024).

[5] K. Martin and B. Hoffman, *Mastering CMake version 3.1*, Kitware: Clifton Park, N.Y. (2015).

[6] C. Scott, *Professional CMake: A Practical Guide.* https://crascit.com/professional-cmake (self-published, 2018–2024).

[7] J. Wuttke, S. Cottrell, M. A. Gonzalez, A. Kæstner, A. Markvardsen, T. H. Rod and G. Vardanyan, *Guidelines for collaborative development of sustainable data treatment software*. J. Neutron Res. **24**, 33 (2022).

[8] M. Stevanovic, *Advanced C and C++ compiling*, Apress: Berkeley (2014).

[39] https://jugit.fz-juelich.de/mlz/bornagain/-/blob/main/AUTHORS.

[9] J. R. Levine, *Linkers and loaders*, Academic Press: San Diego (2000).

[10] D. M. Beazley, B. D. Ward and I. R. Cooke, *The Inside Story on Shared Libraries and Dynamic Loading*. Comput. Sci. Eng. **3**, 90 (2001).

[11] U. Drepper, *How To Write Shared Libraries.* https://www.akkadia.org/drepper/dsohowto.pdf (version 4.1.2, published on the author's homepage, 2011).

[12] G. van Rossum and the Python development team, *Extending and Embedding Python, Release 3.12.2*. Python Software Foundation. https://fossies.org/linux/python-docs-pdf-a4/extending.pdf (2024).

[13] K. Hinsen, *Technical Debt in Computational Science*. Comput. Sci. Eng. **17**, 103 (2015).

[14] M. von Löwis, *PEP [Python Enhancement Proposal] 384 — Defining a Stable ABI.* https://peps.python.org/pep-0384.

[15] B. Peterson, *PEP [Python Enhancement Proposal] 387 — Backwards Compatibility Policy.* https://peps.python.org/pep-0387.

[16] D. Holth, *PEP [Python Enhancement Proposal] 427 — The Wheel Binary Package Format 1.0.* https://peps.python.org/pep-0427.

[17] *Python Packaging User Guide.* https://packaging.python.org.

[18] D. Holth and S. Bidoul, *PEP [Python Enhancement Proposal] 660 — Editable installs for pyproject.toml based builds (wheel based).* https://peps.python.org/pep-0660.

[19] N. J. Smith and T. Kluyver, *PEP [Python Enhancement Proposal] 600 — Future 'manylinux' Platform Tags for Portable Linux Built Distributions.* https://peps.python.org/pep-0600.

[20] R. Jones, *PEP [Python Enhancement Proposal] 301 — Package Index and Metadata for Distutils.* https://peps.python.org/pep-0301.

[21] D. Ingram, *PEP [Python Enhancement Proposal] 566 — Metadata for Python Software Packages 2.1. https://peps.python.org/pep-0566.*

[22] B. Stroustrup, *Learning and Teaching Modern C++.* https://www.youtube.com/watch?v=fX2W3nNjJIo (talk at CppCon 2017).

[23] T. Peters, *PEP [Python Enhancement Proposal] 20 — The Zen of Python.* https://peps.python.org/pep-0020.