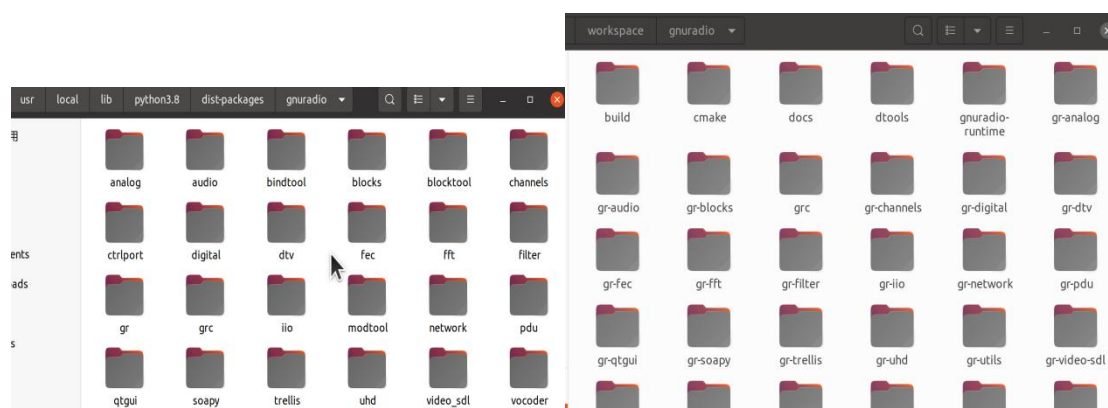


GNU Radio(UHD)整体编译流程以及库依赖

从 git 下 GNU Radio 的官方仓库，到 `sudo make install` 完 GNU Radio 结束，我们实现了在 python 中调用 GNU Radio 库如同普通 python 库一样的操作。但是源代码杂糅了 C++ 和 python 文件的编写，比较晦涩难懂，各种函数的依赖较为复杂，这里我将结合实际阅读源代码的经验出发，简明介绍 GNU Radio 库的整体编译流程。UHD 类同。

Gnuradio 在编译安装后，会在 `/usr/local/lib/python3.8/dist-packages` 下安装 gnuradio 对应的 python 库并允许在编程中调用。查看该文件夹下的具体内容，可以发现其结构与 git 下来项目文件的结构很相似，如下图所示。实际上，库函数的每个文件夹都是对应的项目工程文件的文件夹编译而来的（这也是为什么称之前的安装方式为源码编译的原因）

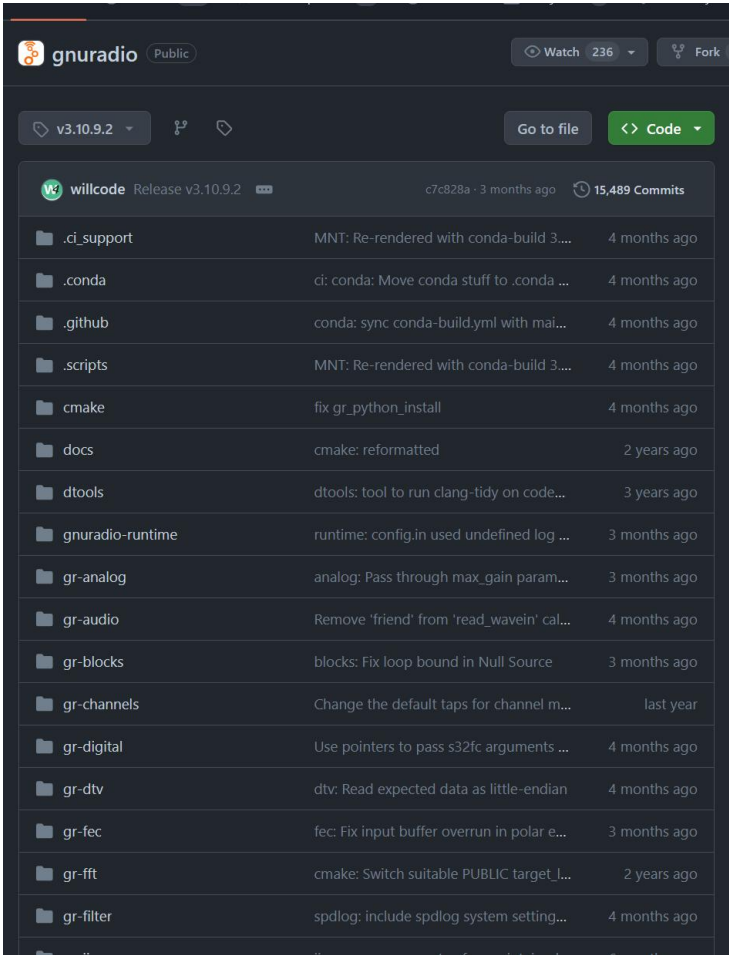


比如，项目工程文件的 `gnuradio-runtime` 被编译后安装为了名为 `gr` 的文件夹中，作为库函数 `gnuradio.gr` 可以在 python 中进行调用。事实上，项目工程中的 `gnuradio-runtime` 文件夹(`gr`)是整个 gnu radio 框架的基石，它包含了 gnu 算法处理过程中的基类库函数 `gr`、`gr.block` 等基础信号块类与传递类这样的基础 class，

并被其他所有的核心模块所调用以搭建 gnu radio 流图。它同时包含了用于 gnu 中的基本数据存储形式之一 pmt 库。另外一个例子是，项目文件夹 gr-digital 编译后=> digital 库，把这个包转换成可以被 python import 的库文件。

接下来详细描述原始项目工程里每个文件夹转换编译为 python 库文件过程中所起的作用，以及实际算法中融合 C++和 python 的方法与联系。

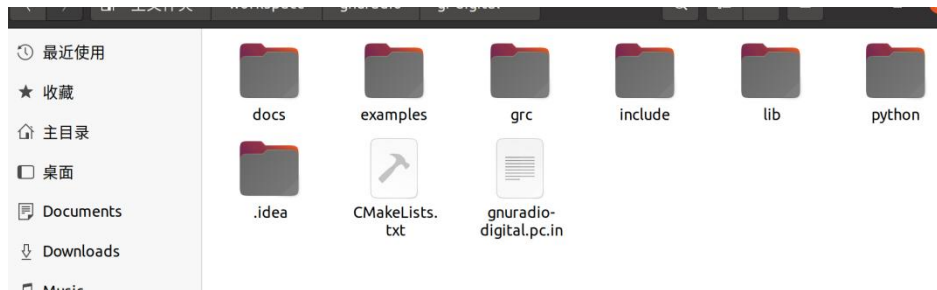
首先 git 完 GNU Radio 的仓库后，原始代码的文件夹如下：



每个文件夹对应一系列通信仿真算法和块。这里以 gr-digital 来举例，我将逐步介绍从创建 build 文件夹开始，到正式编译出可用的 gnuradio 库后，在这个过程中该文件夹下的每项内容是如何——对应到 grc 框图中，以及用 C++编

写的算法是如何被调用于 python 环境中并与 python 一起撰写的算法协同进行信号处理的。

最初的 gr-digital 文件夹下的内容如下：

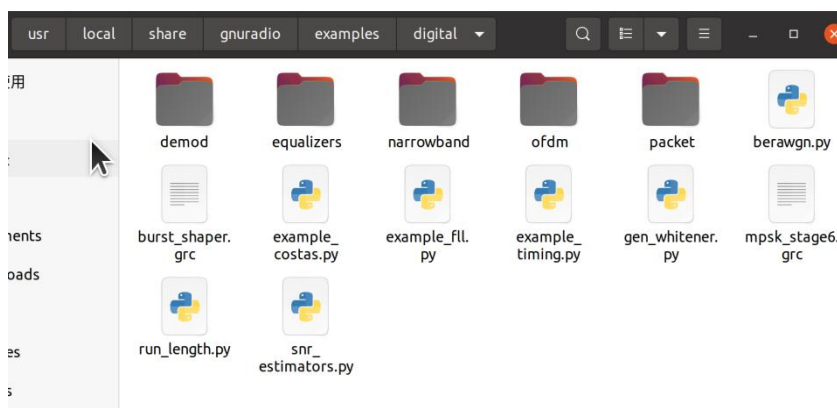


● docs 文件夹

该文件夹包含了有关该模块的文档帮助信息，并包含一个 cmakefiles 文件夹，该文件的作用是 install digital 为一个 python 库函数时，能够通过调用 python 中的 help 方法来获取该模块的帮助信息。

● Examples 文件夹

该文件夹则是包含了一系列作者编写的使用例程，以帮助使用者快速熟悉该库函数的调用、使用方法以及对应 grc 框图的一些基本例程。该文件夹在 cmakefile 文件夹指导下被 make 编译安装到如下 usr/local/share 路径中（该路径通常包含一些与架构无关的共享数据，如文档、图标、字体等。）：



(拓展: ../share 文件夹的作用: 顺便看看 linux 文件系统中各个文件夹的基本用

途:



我观察后发现, 在linux Ubuntu系统中的usr目录下文件夹和usr/local目录下的文件夹命名分类有一定相似之处, 请你帮我解释一下每一个文件夹的作用



ChatGPT

在Linux系统中, `usr` 和 `usr/local` 目录包含了系统的许多核心组件。它们的主要区别在于, `usr` 通常由系统的包管理器管理, 并包含了系统默认的程序和库文件, 而 `usr/local` 通常用于存放系统管理员手动安装的软件, 以避免覆盖或冲突与包管理器安装的文件。

下面是各个目录的一般用途:

1. `usr/bin`: 这个目录包含了用户可执行的二进制程序。在大多数Linux发行版中, 当你安装一个软件包时, 它的可执行文件大多数情况下会被放在这里。
2. `usr/etc` (如果存在): 某些Linux系统可能会在这个目录下存放配置文件, 但这不太常见。通常, 全局配置文件位于 `etc` 目录下。
3. `usr/games`: 专门用于存放游戏程序的目录。
4. `usr/include`: 这个目录包含了C/C++编程语言的头文件, 这些文件在编译程序时需要。
5. `usr/lib`: 包含程序运行所需的库文件。这些库是许多程序共享的代码段, 提供一些通用的功能。
6. `usr/sbin`: 包含一些系统管理的二进制程序, 通常只有root用户才会使用到。
7. `usr/share`: 这个目录包含了与架构无关的共享数据, 如文档、图标、字体等。
8. `usr/src`: 包含源代码, 例如Linux内核的源代码可能就放在这里。

在 `usr/local` 目录下, 你可能会发现类似的子目录:

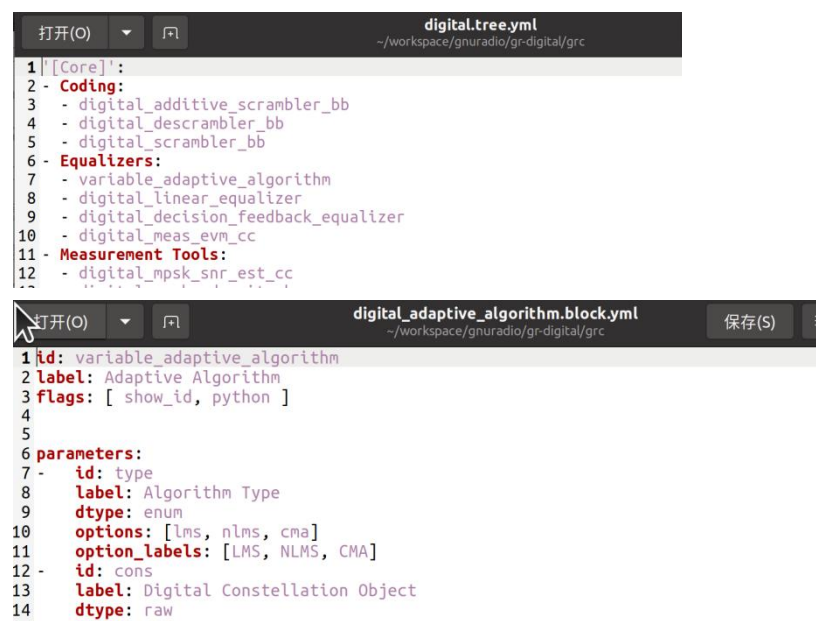
1. `usr/local/bin`: 存放系统管理员手动安装的用户可执行程序。
2. `usr/local/etc`: 如果存在, 可能会用于存放手动安装的软件配置文件。
3. `usr/local/games`: 存放手动安装的游戏程序。
4. `usr/local/include`: 存放手动安装软件的C/C++头文件。
5. `usr/local/lib` 和 `usr/local/lib64`: 包含手动安装的库文件, `lib64` 专用于64位库。
6. `usr/local/sbin`: 存放手动安装的系统管理程序。
7. `usr/local/share`: 存放手动安装软件的架构无关数据。
8. `usr/local/src`: 存放手动安装的软件源代码。

使用 `usr/local` 目录来安装软件可以避免与由系统包管理器安装的程序混淆, 使得管理员手动安装的软件更易于管理。同时, 这样做也遵循了文件系统层次标准 (Filesystem Hierarchy Standard, FHS), 这是大多数Linux发行版遵循的目录组织标准。 ↓

接下来的几个文件夹是算法的核心内容。

● grc 文件夹

该文件夹是存放可视化 gnu radio companion 框图的配置的地方。该文件夹由一个 tree 文件和一系列不同算法框图配置文件组成, tree 文件列出了本模块包含的所有 grc 框图架构以及对应分类名称, 具体的算法 grc 框图配置文件则以规范的方式命名了该算法在 grc 中显示的名字和对应输入输出接口的命名和框图里参数的具体配置参数:



```
1 ['Core']:  
2 - Coding:  
3   - digital_additive_scrambler_bb  
4   - digital_descrambler_bb  
5   - digital_scrambler_bb  
6 - Equalizers:  
7   - variable_adaptive_algorithm  
8   - digital_linear_equalizer  
9   - digital_decision_feedback_equalizer  
10  - digital_meas_evm_cc  
11 - Measurement Tools:  
12  - digital_mpsk_snr_est_cc  
13  
14
```

```
1 id: variable_adaptive_algorithm  
2 label: Adaptive Algorithm  
3 flags: [ show_id, python ]  
4  
5  
6 parameters:  
7 - id: type  
8   label: Algorithm Type  
9   dtype: enum  
10  options: [lms, nlms, cma]  
11  option_labels: [LMS, NLMS, CMA]  
12 - id: cons  
13   label: Digital Constellation Object  
14   dtype: raw
```

在 make install 编译安装后, 该 grc 文件夹同样被存放在 usr/local/share 路径中, 最终的运行 grc 时会通过 gnuradio.grc 库中的代码与实际模块命名对应的算法联系在一起, 完成对信号的处理并可视化于 grc 界面中。

● Include 和 lib 文件夹

如同 C++项目命名方式一样, 这两个文件夹下包含了使用 C++代码实现算法的源码。

- 其中 include 文件夹包含了所有使用 C++ 算法编写的代码中函数的声明, 或者说提供了 api 接口, 在应用时, 如果要使用这些 library, 只需要 include 这些头文件即可。在 make 编译安装后, 这些 include 文件夹被自动添加到 linux 系统目录 /usr/local/include 下, 例如, 对于 digital 包, 我们可以在 “/usr/local/include/gnuradio/digital” 下找到这些头文件, 这些头文件与 git 下源代码仓库里的 include 内容一样。同理对于 sudo make install 的其他库 (uhd, volk), 都可以在这里找到。(补充说明, C++ 自带的库可以在 /usr/include/ 目录下找到)
- Lib 文件夹则包含了原始算法函数的具体 C++ 实现源代码, 在 git 下来的项目工程文件中展示, lib 文件夹里有 .h 和 .cc 后缀的文件, 这里 .h 是进一步实现函数时会用到的一些声明, 但是需要与刚才 include 文件夹里的 .h 文件作区分, 区别就是, lib 里的 .h 只供内部 .cc 使用, 不对外暴露接口公共使用, 或者说, lib 文件夹里的 .h 文件辅助这里的 .cc 文件 (也就是具体函数实现代码) 编译(非公共使用); 而 include 文件夹里的 .h 是会最终对外提供 api 的接口(公共使用)。在 make 编译安装时, 会结合 include 和 lib 这两个文件夹, 将 lib 中的具体实现算法编译生成链接库文件, 这些文件可以在 usr/local/lib 下看到, 如下图所示, 诸如 libgnuradio-analog.so 即为这些编译好的动态链接库文件, 这些文件包含了源项目工程中的 lib 下 C++ 实现的算法。在源代码 .cc 编译为动态链接库后, 可以直接通过调用 usr/local/include 中的头文件中声明的函数(api)来调用这些算法。



一些补充的扩展阅读资料：

在 C++ 工程结构中，include 和 lib 文件夹具有特定的作用：

1. include 文件夹：

这个文件夹主要用来存放头文件（.h 或 .hpp），头文件中包含了函数声明、类定义、宏定义以及其他预处理器指令等。当开发 C++ 程序时，编译器需要访问这些头文件来获取程序所需的类型、函数接口等信息。例如，如果你正在使用某个第三方库，其 API（应用程序编程接口）通常通过包含相应的头文件来引入到你的源代码中。

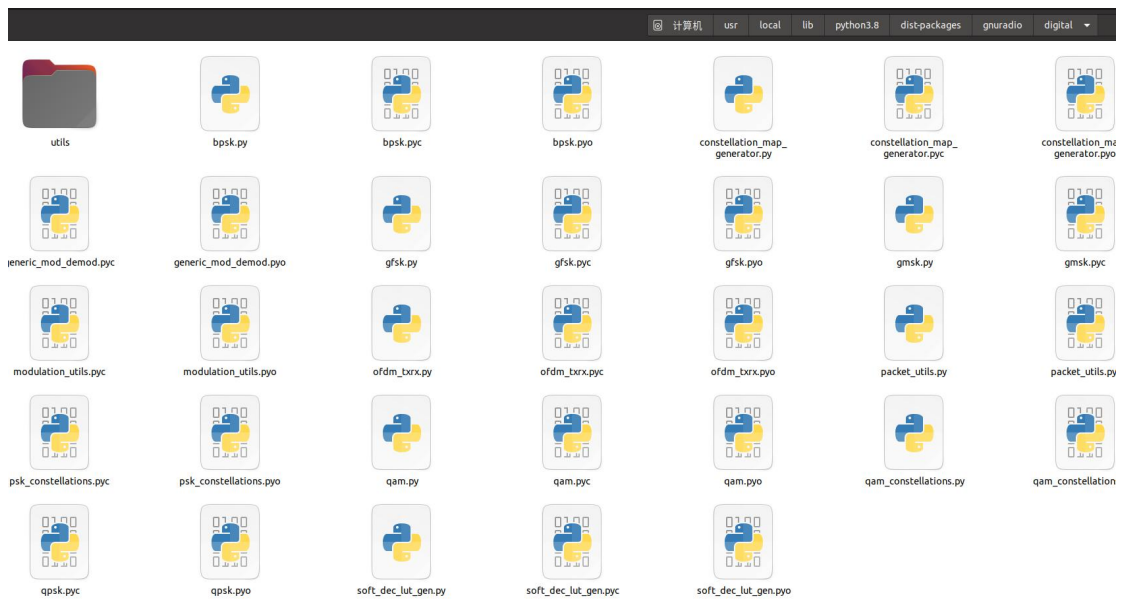
2. lib 文件夹：

这个文件夹主要用于存放库文件，通常是静态库（.lib 文件在 Windows 平台下）或动态库（.dll 在 Windows 下，.so 在 Linux 下，.dylib 在 MacOS 下）。库文件包含了已经编译好的可重用的二进制代码，它们实现了在头文件中声明的功能。在链接阶段，编译器或链接器会查找并链接这些库文件，将它们所含的函数和对象整合到最终生成的可执行文件中，这样程序运行时就可以调用库中的功能。

总的来说，include 文件夹是用来包含源代码编译时所需的信息，而 lib 文件夹则是用来存放编译后供链接阶段使用的库文件，二者共同协作使得程序能够正确编译和运行时调用库提供的功能。在实际的开发环境中，如 Visual Studio，为了方便管理与复用，开发者常常需要在项目设置中指定这两个路径，以便编译器和链接器能自动找到对应的头文件和库文件。

● Python 文件夹

原工程文件中的 python 文件夹是将 gnuradio 变为易于在 python 中以库的形式调用的核心文件夹。Python 文件夹的结构为，这里以 gr-digital 为例：python----digital（去掉 gr 的库文件名称，也就是最终在 python 中展示的或者被调用时引用的库的名称）-----bindings 文件夹+一系列 python 源文件。先说一系列 python 源文件。这里的 python 源文件由两个部分组成：第一个是以功能命名（比如 digital 里的 bpsk.py ofdm_txrx.py __ini__.py 等）的.py 文件，这些文件是 python 库里的核心函数，当在 python 中 import 该库时，就通过使用这些文件提供的方法，类，函数来构建算法。第二个部分是以 qa_开头命名的.py 文件，这里 qa 的意思是 qualify assurances,即质量保证，这些文件在 make 编译后会通过 make test 命令测试，即当时 test 的源文件就来自这里，这些文件是基于第一部分中的基础功能函数文件，从而实现了某个特定的算法（比如 qa_crc16_async_bb.py），这些算法最后被用来 make test 以验证 gnuradio 编译是否正确，功能是否正常。最后经过编译后，这些 python 文件被打包成标准 python 库函数，可以在代码中 import，这些库函数可以在 /usr/local/lib/python3.8/dist-packages/gnuradio 目录下查看，具体结构（这里以 digital 库举例）如下图所示。可以看到这些库函数文件与源工程文件 python 文件夹下的第一部分内容跟一致。



这里最核心的部分，也就是真正融合 C++和 python 实现的方法便是库文件里



digital_python.cpython-38-x86_64-linux-gnu.so

的 文件，这个文件是通过 pybind11 库来将 C++实现的算法块转化为可以被 python 调用并在 python 环境中使用的库函数，调用方法如下图所示。

```
import numpy

from gnuradio import gr, blocks, fft, analog

from . import digital_python as digital
```

from . import digital_python as digital 这一行将该动态链接库.so 文件转化为了 python 库文件，并在 python 环境中使用该库中由 C++实现的算法。

在 python 库中的这个.so 文件是通过 bindings 文件夹，也就是前文中提到的

项目源代码中 python 文件夹下的一个子文件夹，来构建的。bindings 的文件夹包含了与 C++ 文件命名方式类似的 .cc 后缀文件。在涉及 C++ 和 Python 混合编程的项目中，`bindings` 目录通常用于存放**绑定代码**。这些绑定代码是使用专门的库（如 pybind11）编写的 C++ 文件，其作用是为 C++ 代码创建 Python 接口，这样编写的 C++ 算法就可以在 Python 环境中被调用和使用。

联系到之前项目工程里的 include and lib 文件夹，`bindings` 目录和项目中的 C++ 代码 `include` 和 `lib` 目录之间联系如下：

1. include/lib 目录： 这些目录通常包含项目的核心 C++ 代码。`include` 目录存放头文件（`.h` 或 `.hpp` 文件），它们声明了类、函数和模板等。`lib` 目录包含 C++ 源代码文件（`.cpp` 文件），并在 make 后被编译为动态链接库文件存放在系统目录中（如 `.a` 或 `.so` 文件）。

2. bindings 目录： `bindings` 目录中的 `.cc` 文件包含用于将之前实现的 C++ 类和函数“绑定”到 Python 可调用的形式的代码。使用 pybind11，这些文件中的代码会映射 C++ 类和方法到 Python 对象和函数。这样，Python 用户就可以像使用原生 Python 模块一样使用这些 C++ 编写的组件。

举例： 如果有一个 `additive_scrambler` 的 C++ 类（在 include 文件夹下命名为 `additive_scrambler_bb.h`），`bindings` 目录下就有一个 `additive_scrambler_bb_py.cc` 的文件，该文件的代码将会使用 pybind11 定义如何在 Python 中创建 `additive_scrambler` 对象以及如何调用其方法。

3. 工程 make 构建过程： 在构建过程中，C++ 编译器会先编译 `include` 和 `lib` 目录下的 C++ 代码，生成动态链接库文件并存储在系统目录中。然后，pybind11 会用 `bindings` 目录下的 `.cc` 文件来生成可以被 Python 导入的扩展模块，即我们之

前图片中在 python 库中看到的`.so`（共享对象）文件。

通过这种方式，`bindings`中的代码充当了 C++和 Python 之间的桥梁，使得 Python 代码能够访问和使用 C++中定义的数据结构和算法。这对于性能关键部分使用 C++编写，而控制和高级逻辑使用 Python 编写的项目特别有用，这样既可以利用 C++的性能优势，又可以享受 Python 的快速开发和易用性。

总而言之，include 和 lib 文件夹中的 C++文件通常是 GNU Radio 项目的核心组件，它们定义了基础通信算法的实现，并且会被编译成库文件供其他 C++代码使用。include 文件夹通常包含头文件，而 lib 文件夹包含源文件和库文件。bindings 文件夹中的 C++文件会依赖 include 文件夹中的头文件和 lib 文件夹中的实现文件（动态链接库文件）。它们通过#include 指令来引用所需的头文件，确保 C++代码的完整性和正确性。

下图展示的就是 bindings 文件夹下的 additive_scrambler_bb_py.cc，可以看到它调用了之前 C++代码结构中 include 文件夹下的头文件，以调用 lib 文件夹中实现对应的算法函数接口。最终该文件将构建 python 库中的.so 文件中的一部分，以在 python 环境下调用该 C++实现的 additive_scramble 算法。

```
#include <pybind11/complex.h>
#include <pybind11/pybind11.h>
#include <pybind11/stl.h>

namespace py = pybind11;

#include <gnuradio/digital/additive_scrambler_bb.h>
// pydoc.h is automatically generated in the build directory
#include <additive_scrambler_bb_pydoc.h>
```