NJOY Documentation

Jeremy Lloyd Conlin

Jan 27, 2021

Contents:

The documentation for NJOY2016 is currently hosted on: GitHub.

Keep this page bookmarked for additional information.

CHAPTER 1

Installing NJOY

Both NJOY21 and NJOY2016 use the same configuration and build process. Additionally, all of the supporting components use the same process. The steps on this page are for building NJOY21, but are equally applicable to NJOY2016.

1.1 For the impatient

```
# Download the latest version of the code
git clone --branch v1.2.1 https://github.com/njoy/NJOY21.git

cd NJOY21

# Configure the build process
mkdir bin
cd bin
cmake -D CMAKE_BUILD_TYPE=Release ..

# Build NJOY21
make

# Test NJOY21
make test
```

1.2 Prerequisites

Development for NJOY uses the latest published language standards that are widely supported by compiler vendors, *at the start of development*. Therefore, the minimum language standards are

- C++17 or higher
- Fortran 2003 or higher

- Python 3.4+

Additionally, we use CMake to configure the build system and git for version control.

- CMake 3.14+
- git 2.1+

1.2.1 Windows

When building on Windows, we only support using the Windows Subsystem for Linux. The build system is the same.

1.3 Build Process

To download NJOY21, simply `git clone` the repository. First move into the directory where you want the source code, then execute:

```
git clone https://github.com/njoy/NJOY21.git
```

This will get the latest version of the code. If you are interested in a particular version, you need to specify the git branch corresponding to that version like this for version 1.2.1:

```
git clone --branch v1.2.1 https://github.com/njoy/NJOY21.git
```

Please note that getting versions of NJOY21 prior to version 1.2.1 was different—and more convoluted. For information on how to do that see this page.

The configuration and build is performed in a directory (`bin`) inside the source directory

```
cd NJOY21
mkdir bin
cd bin
cmake -D CMAKE_BUILD_TYPE=Release ../
```

Note this will require a connection to the internet as `cmake` command will download the necessary dependencies.

```
make
```

You can provide the command-line option `-j n` (where `n` is the number of concurrent/parallel "jobs") to speed up the compilation step.

```
make test
```

Like for the compiling step, you can pass the `-j` command-line option here as well.

If all the tests pass, you should see something that looks like:

```
100% tests passed, 0 tests failed out of 90
```

For this configuration/build process, a connection to the internet is required as subprojects are downloaded from GitHub. There are many instances when one would need to build NJOY on a machine that is not connected to the internet. To do that, please use these steps:

```
# Download the source code
git clone https://github.com/njoy/NJOY21.git

# Configure the build process
cd NJOY21

#### Transfer the entire NJOY21 directory to machine
#### not connected to the internet

cd NJOY21
mkdir bin
cd bin
cmake -D CMAKE_BUILD_TYPE=Release ..

# Build NJOY21
make

# Test NJOY21
make test
```

1.3.1 Compliant Compilers

We have tested NJOY2016 and NJOY21 with the following compilers.

C++17 Compliant Compiler

We *strongly* suggest building with the LLVM C++ compiler. We have found this compiler to be *much* faster in compiling as well as being fast at runtime.

- **Mac:**
  - LLVM 9.0.1. The version that comes with XCode will be sufficient.
- **Linux:**
  - gcc 8.3

  Note that in Windows we currently support compiling and running under the Windows Subsystem for Linux

Fortran 2003 Compliant Compiler

- **Linux:**
  - gfortran This is included as part of the standard gcc suite of compilers.
- **Mac:**
  - Installing gcc 8.3 via homebrew or spack works quite well.

Other compilers will most likely compile without any problems, but may give different answers. The differences are typically small and due to different ways of optimizing the code from each compiler. The differences are not likely to be significant.

CHAPTER 2

Legacy Input

The input for Legacy[1] NJOY begins with the case-insensitive name of the module that is to be run. Following the module name, various "cards"[2] are listed with each card having multiple arguments. Each card is terminated with an optional highly recommended) forward slash `/` followed by a carriage return. Any text following the forward slash is ignored and thus can be used as a comment.

Tip: It is highly recommended that you terminate every Card with a forward slash—even though it is not required. This forces NJOY to stop looking for more input for that Card. It also helps the writer of the input to know where things end.

The last line of the input is the word `stop`, which tells NJOY to stop processing. The input would look something like this:

```
MODULE1
   arg1 arg2 arg3 / Card1
   arg4 arg5      / Card2
   ...
MODULE2
   arg1 arg2      / Card1
stop
```

The precise definitions of the arguments and cards for each module can be found in the NJOY2016 manual. Since NJOY21 uses the same input as Legacy NJOY, we include some documentation here.

2.1 General Information

There are a few input arguments that are common among all/most modules that are worth explaining here.

[1] Legacy NJOY refers to the Fortran-based NJOY, namely, NJOY2016 and its predecessors.

[2] The term card refers to the physical punch cards used in (very) old computer codes.

comments Lines that begin with two hyphens followed by a space, `--`—and that occur *before* the module name—are ignored by the input parsers. These lines can be used as comments. Also, everything after the forward slash terminating a Card is ignored and can also be treated as a comment.

input/output unit Most modules have arguments specifying the input and/or output "tapes". The argument is an integer, with a number between 20 and 99 inclusive. NJOY will read/write from/to actual files with the name `tape##` where `##` is the value of the input/output unit. These tape files must be in the same directory from which NJOY was called. Negative numbers refer to binary tapes and positive numbers refer to ASCII tapes.

CHAPTER 3

NJOY21 Components

This page serves to document the various components that make up NJOY21.

3.1 NJOY Modules

The biggest components of NJOY21 are called *modules*. These modules are those that replicate the functionality of Legacy[1] NJOY modules. Listed below are the modules that have been modernized and updated. The documentation for each is given in their respective locations. If a module is not listed here, please see the documentation for Legacy NJOY.

RECONR Resonance reconstruction

Thermal Scattering Introduction to thermal scattering, LEAPR, and THERMR.

3.2 Modern NJOY Components

There are many components that make up NJOY21, or that make up one of it's modules. These components can be used independently or integrated with others. Listed here are just a few of the components. All of our components can be found on our GitHub page: https://github.com/njoy.

ENDFtk Toolkit for working with ENDF-formatted files.

GNDStk Toolkit for working with GNDS-formatted files.

resonanceReconstruction Perform resonance reconstruction calculations for these formalisms

- Single-level Breit-Wigner,
- Multi-level Breit-Wigner,
- Reich-Moore,
- R-Matrix Limited.

[1] Legacy NJOY refers to NJOY2016

resonanceReconstruction can also reconstruct unresolved resonances from the following ENDF formats:

- Energy-independent (Case A),
- Energy-independent Fission Widths (Case B),
- Energy-dependent (Case C).

interpolation Library for interpolating one-dimensional data.

CHAPTER 4

For Developers

Please be patient as this is a work in progress.

4.1 Integration of Modern NJOY21 Component

When a new component (i.e., module) is modernized and added to NJOY21, we need to tell NJOY21 how to link to the component and what the new component needs to accept as arguments so that it can do what it needs to do.

In this document, I use the THERMR component as an example.

4.1.1 Summary

In summary here are the steps that need to be done to integrate a new component into NJOY21. Each of these steps is covered in detail in this document.

1. Teach `lipservice` how to convert input arguments to JSON, e.g., `THERMR`
2. Remove module name (`THERMR`) from `setupLegacyDirectory` call.
3. Add module name (`THERMR`) to `setupModernDirectory` call.
4. Define modern "sequence" routine for new component.
5. Create `THERMR` class (in THERMR repository) with call operator, `operator()`, taking two `nlohmann::json` arguments.
6. Add new component (i.e., THERMR repository) as git submodule to NJOY21.

4.1.2 1. Passing input arguments to modern component

Let's look at a typical input for THERMR:

```
thermr
  0 -22 -24 /
  0 1306 8 2 1 0 0 1 221 2 /
  350.0 450.0 /
  0.05 1.2 /
```

In NJOY21, the input deck is read by lipservice. In order to pass the arguments to the modern component we have to convert them into a JSON object. (Arguments are passed to legacy modules in some other magical fashion.) The JSON object for this input looks like:

```
{
  "nendf":  0,
  "nin":    -22,
  "nout":   -24,
  "matde":  0,
  "matdp":  1306,
  "nbin":   8,
  "ntemp":  2,
  "iin":    1,
  "icoh":   0,
  "iform":  0,
  "natom":  1,
  "mtref":  221,
  "iprint": 2,
  "tempr":  [ 350.0, 450.0 ],
  "tol":    0.05,
  "emax":   1.2
}
```

A few things to note:

- The keys are the same argument names as in the NJOY2016 documentation (and likely the code).
- There are no "cards" in the JSON object.
- Notice that the temperatures are turned into a JSON array `[ 350.0, 450.0 ]`.

Conversion from `lipservice` to JSON

We are using the excellent JSON library by Niels Lohmann for C++ JSON capabilities. That library provides abilities to convert custom objects into a JSON object. In the `lipservice` repository is demonstration of how this is done for the THERMR module in file (shown below).

Note that there is a `to_json` function for each THERMR card and lastly a `to_json` function for the `lipservice::THERMR` object itself.

```
inline void to_json( nlohmann::json& JSON, const THERMR::Card1& card1 ) {
  JSON = {
    { "nendf",  card1.nendf.value },
    { "nin",    card1.nin.value },
    { "nout",   card1.nout.value }
  };
}

inline void to_json( nlohmann::json& JSON, const THERMR::Card2& card2 ) {
  JSON = {
    { "matde",  card2.matde.value },
```

(continues on next page)

(continued from previous page)

```
    { "matdp",  card2.matdp.value },
    { "nbin",   card2.nbin.value },
    { "ntemp",  card2.ntemp.value },
    { "iin",    card2.iin.value },
    { "icoh",   card2.icoh.value },
    { "iform",  card2.iform.value },
    { "natom",  card2.natom.value },
    { "mtref",  card2.mtref.value },
    { "iprint", card2.iprint.value }
  };
}

inline void to_json( nlohmann::json& JSON, const THERMR::Card3& card3 ) {
  JSON = { { "tempr", card3.tempr.value } };
}

inline void to_json( nlohmann::json& JSON, const THERMR::Card4& card4 ) {
  JSON = {
    { "tol", card4.tol.value },
    { "emax", card4.emax.value }
  };
}

inline void to_json( nlohmann::json& JSON, const THERMR& thermr ) {
  JSON = thermr.card1;
  JSON.update( thermr.card2 );
  JSON.update( thermr.card3 );
  JSON.update( thermr.card4 );
}
```

This all goes in `src/lipservice/src/THERMR.hpp` in `lipservice`. An include statement needs to be added to `src/lipservice/src/to_json.hpp`

```
#include "lipservice/src/THERMR.hpp"
```

4.1.3 2. Modifying routines to set legacy and modern directories

When NJOY21 executes, it sets up some "directories" (these are *not* the same as folders on your computer), a directory for the legacy modules and a directory for the modern components. It does this in the `setupLegacyDirectory` and `setupModernDiretory` respectively.

In file `src/njoy21/Driver/Factory/src/setupLegacyDirectory.hpp`:

```
static Directory setupLegacyDirectory( CommandLine& commandLine ){
  return ( commandLine.legacySwitch ) ?
    Directory( { "MODER", "RECONR", "BROADR", "PURR", "UNRESR", "ACER",
                 "GASPR", "HEATR", "GROUPR", "VIEWR", "MIXR", "DTFR",
                 "THERMR", "LEAPR", "RESXSR", "MATXSR", "GAMINR", "PLOTR",
                 "COVR", "WIMSR", "POWR", "CCCCR", "ERRORR" } ):
    Directory( { "MODER", "RECONR", "BROADR", "PURR", "UNRESR", "ACER",
                 "GASPR", "HEATR", "GROUPR", "VIEWR", "MIXR", "DTFR",
                 "LEAPR", "RESXSR", "MATXSR", "GAMINR", "PLOTR",
                 "COVR", "WIMSR", "POWR", "CCCCR", "ERRORR" } );
}
```

Please note that `"THERMR"` occurs in the first `Directory` function call, but not in the second. **When a modern component is added, the name of the new component needs to be removed from this second `Directory` call.**

In file `src/njoy21/Driver/Factory/src/setupModernDirectory.hpp`:

```
static Directory setupModernDirectory( CommandLine& commandLine ){
  return ( commandLine.legacySwitch ) ?
    Directory() :
    Directory( { "THERMR" } );
}
```

When a modern component is added, the name of the new component needs to be added to this second `Directory` call.

4.1.4 4. Define a modern "sequence"

This part is really easy. In file `src/njoy21/modern/Sequence/routines.hpp` just add the line `DEFINE_ROUTINE( THERMR )` at the end of the file (but before the line `#undef DEFINE_ROUTINE`).

```
#define DEFINE_ROUTINE( MODULE )                                                  \
  struct MODULE : public interface::Routine {                                     \
    nlohmann::json j##MODULE;                                                     \
                                                                                  \
    template< typename Char >                                                     \
    MODULE( lipservice::iRecordStream< Char >& stream ){                          \
      lipservice::MODULE command( stream );                                       \
      this->j##MODULE = command;                                                  \
    }                                                                             \
    void operator()( std::ostream& output,                                        \
                     std::ostream& error,                                         \
                     const nlohmann::json& args ){                                \
      njoy::MODULE::MODULE{}( std::move( this->j##MODULE ),                       \
                              output, error,                                      \
                              args );                                             \
    }                                                                             \
  };

  DEFINE_ROUTINE( THERMR )
#undef DEFINE_ROUTINE
```

4.1.5 5. Create a functor for the component

In `src/njoy21/modern/Sequence/routines.hpp` we can see (although it's a bit obscure) that when NJOY21 calls the new component, it does so by calling the call operator on an object with the same name as the new component.

```
class THERMR {
public:
  void operator()( const nlohmann::json& njoyArgs,
                   std::ostream& output,
                   std::ostream& error,
                   const nlohmann::json& args ){
    // Do something here
  }
};
```

This call operator does everything that needs to be done for the modern component. The `njoyArgs` argument is the JSON object that was created by lipservice as described earlier. To access the different members, simply use the bracket operator:

```
njoyArgs[ "iprint" ];
```

What is done with the parameters in `njoyArgs` is up to whoever wrote the component.

The `output` argument is where messages about the processing should be "printed". In Legacy NJOY, this would be the `output` file. In NJOY21, the name of this file is specified using the `-o` or `--output` command-line argument.

The `error` argument is similar to the `output` argument, except that error messages are written here. Error messages are messages written before NJOY crashes (hopefully gracefully).

The `args` argument is another JSON object that contains an arbitrary set of options that are passed to every modern component. Right now this is just a placeholder for unknown future needs.

When that function finishes, NJOY21 continues by running the next module given in the input deck, so everything that needs to be done with the modern component must be taken care of in this call operator. Likely, this function just calls other functions to do the real work.

4.1.6 6. Adding new component as submodule to NJOY21

Once the new component has been implemented (in a separate repository) it needs to be added to NJOY21.

Since we are currently improving our build system, the steps for this are still being developed.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Index