X
User Documentation

Mohammed Rahman

February 11, 2025

Version 2.1

Contents

1 Introduction

1.1 Overview

X is a powerful software designed to [brief description of what your software does]. This document provides a comprehensive guide on how to install, configure, and use X effectively.

1.2 Features and use cases

2 System Requirements

Before proceeding, ensure that your system is running HPC Ubuntu, as this software has not currently been tested on other systems.

3 Quick Start

Follow these steps to quickly set up and run the software. For detailed instructions, see the following sections.

3.1 Spack Envrionment

3.1.1 First-Time Spack Setup

```
1     git clone {INSERT_SSH_LINK}
2     cd spack_area
3
4     spack repo add name
5     s-e
6
7     spack env activate -p name # If successful, the environment name
          should be visible
8     spack add package
9     spack concretize -p name
```

3.1.2 Load Existing Spack Environment

```
1     spack env activate -p name # If successful, the environment name
          should be visible
```

3.2 Clone the Repository

```
1     git clone {INSERT_SSH_LINK}
2     cd X
```

3.3 Set Envrionment Variables

```
export MKL_DIR=
```

3.4 Build And Install The Software

```
cmake --preset production
cmake --build --preset production
cmake --install build/production
```

3.5 Run Unittests

```
cd build/production
ctest && cd ../..
```

4 Spack Envrionment

This section will guide you through creating the Spack environment and how to reuse it.

To build X, we must first ensure that all required dependencies and specific module versions are available. This is where Spack comes in. X currently supports `Python 3.11.6`, which is managed through Spack.

Spack was chosen to eliminate the need for users to manually install dependencies. Once the environment is set up, it can be easily loaded whenever needed, streamlining the build process. Specifically, Spack loads Python, all its required modules, as well as the GCC compiler.

Note: The Intel compiler is also supported by X; however, it is unstable with Spack and will therefore not be covered in this manual. If a user wishes to use the Intel compiler, they may do so, as the build system will automatically detect Intel and set the appropriate compiler flags accordingly.

4.1 First-Time Spack Setup

When setting up the Spack environment for the first time, we need to clone the Spack repository and add it to our Spack repository list. This ensures that Spack can locate package recipes, which define the dependencies required for the X build.

```
git clone {SPACK_SSH}
load spack # load the spack module
spack repo add <path-to-repo>
```

If the repository is added successfully, Spack will display a confirmation message. To verify that the repository has been added, use the following command:

```
spack repo list
```

The newly added repository should appear at the top of the output. This indicates that Spack will prioritise searching this repository before any others when resolving package dependencies.

Next, we will create an empty environment to manage the dependencies for X:

```
spack env activate -p X
```

The environment name should appear on the left side of the command line prompt. This indicates that the environment is loaded and active, ready for use.

However, the environment is currently empty and not useful. We will now install the X dependencies from the repository we cloned and added. For reference, you can view the specific dependencies by navigating to `packages -> x -> package.py`.

```
spack add X  # Add the X package to our environment
spack concretize -f  # Resolve dependencies and finalise the
   environment setup
```

The concretize step may take some time, but don't worry—it hasn't crashed. Once the concretizing step is complete, this means our environment now contains all the dependencies needed for X and is fully set up.

Note: You only need to perform this entire setup process once. After setup, you can easily load the Spack environment for future use, even when logging back in, as demonstrated in the next section.

4.2 Load Existing Spack Environment

If you have already gone through the process of setting up a Spack environment, you can load it back up whenever needed. Fortunately, this is an easy and fast process!

```
load spack # load the spack module
spack env activate -p X # load the existing envrionment
```

As shown in the commands above, all that is needed is to load Spack and then the environment. If successful, the environment name should appear on the left-hand side of the command line.

4.3 Update Spack Environment

If the Spack environment for X you are currently using is outdated, you will need to update it. This is straightforward—first, pull the updated changes to the Spack repository:

```
cd spack-repo
git pull # update repo with new changes
```

Once the repository has been updated, load the X environment and run the following command:

```
spack concretize -f
```

Now, your environment should be updated and ready to use!

5 CMake Build System

This section will guide you through the build system of X, covering building, testing, and installation. X uses CMake to simplify and streamline this process, ensuring an efficient and consistent workflow.

5.1 Clone X

Firstly, clone X from the Git repository as shown below:

```
git clone {link} --recurse-submodules  # Clone X and pull the
    contents of its submodules
cd X
```

5.2 Build Presets in X

X supports two predefined build presets: **Debug** and **Production**. These presets simplify and streamline the configuration process, allowing you to set up the build environment quickly.

5.2.1 Production Build Preset

The **Production** build preset is optimized for creating the final, production-ready version of X. It includes configurations that enhance build efficiency, reduce build times, and improve runtime performance. This preset is ideal for preparing the software for deployment or distribution and will be the most commonly used by end users.

5.2.2 Debug Build Preset

The **Debug** build preset is designed for development and debugging. It includes settings that enable debugging symbols, optimisations for debugging, and verbose debug outputs. This preset is most likely not going to be used by you, the user, but is still available for both developers and users.

5.2.3 Setting the Build Preset

Once you have selected your desired build preset, you can apply it using the following CMake command. Here, we will use the **production** preset, as it is intended for most users:

```
cmake --preset production
```

This command configures the build system using the settings defined in the selected preset, ensuring an optimised build environment. A build directory will be created during the process, where all produced files will be stored. This is done to keep the source directory clean and avoid clutter.

5.2.4 Preset Flags

5.3 Build X

Once the build preset has been set, the following command can be used to build X using CMake:

```
cmake --build --preset production # either production or debug,
    depending on preset applied
```

This command will build X, more specifically, it will compile and create all the binaries and libraries needed for X. These files will be placed under the build directory, keeping the source directory clean. The process can take a minute or two depending on your system's resources.

You should see two module files generated in the X directory: one named X and another named Dakota. The X module file sets the necessary environment variables, allowing X to be used anywhere on the system without requiring to be in the install directory. Similarly, the Dakota module file ensures that Dakota can be accessed from any location.

Once this step is complete, X is officially built and ready for the installation process.

5.4 Unit Testing

Before installing X, it is highly recommended to run the unit tests integrated within X. These tests can be found in the `X/test` directory. Running the unit tests will provide a good indication of whether the software has been built correctly.

`ctest` is used to run the unit tests, as shown in the commands below:

```
cd build/production  # Use build/debug for a debug build
ctest
cd ../..  # Return to the original directory
```

Running the unit tests should take a minute or two. Once completed, if all tests pass, you're good to go!

If not, you can check `X/build/production/Testing/LastTest.log` to see why the tests are failing. This is typically due to incorrect configuration of the Spack environment.

Note: Adding unit tests for custom models and tools in X is explained in the plugin integration section.

5.5 Install X

Now that all the required libraries and binaries for X have been built, we can install X into the `$install_dir`, which is either user-defined or set to a default location if not specified. Typically, the default installation path is `x/build/production/local_install`.

The installation process involves transferring all required files—and only the necessary ones—needed to run X into a designated directory. This includes source files and essential generated files from the build process. The purpose of this step is to keep the source directory clean while providing a single location where X can be used efficiently.

To install X, simply use the following CMake command:

```
cmake --install build/production  # Use build/debug for a debug build
```

On the command line, you should see all relevant files being copied into the install directory. Once this process is complete, X is officially installed.

5.6 Rebuild X

If you need to rebuild and install X from the beginning for any reason, this can be easily done using CMake, as shown below:

```
    rm -rf build # remove the build directory
    ...  # Commands to build and install X
```

Removing the build directory essentially restarts the entire build process. Since CMake generates files based on the build directory, if it is empty, the build process will start from scratch.

This also removes the install directory set by the default installation path. If a different installation path was used, you will need to remove it manually.

5.7 Summary

As showcased, building X is a straightforward process thanks to CMake. Below are all the commands tied together in order:

```
git clone {link} --recurse-submodules && cd X  # Clone X and pull
    the contents of its submodules

cmake --preset production # set cmake preset
cmake --build --preset production # build selected preset

cd build/production
ctest && cd ../.. # run unit tests

cmake --install build/production # install X into the
    $local_install directory
```

Once installed, X can be used independently or interfaced with Dakota, as shown in the Usage section.

6 Plugin Integration

6.1 Drag-and-Drop System

X is highly flexible when it comes to working with different models, tools, and integrators. Thanks to CMake, all models, integrators, and most tools (apart from k) can be removed and interchanged freely with minimal effort.

This is evident in the `plugin` directory. All models, tools, and integrators present in this directory can be removed by simply dragging them out of the directory, and added by dropping a new model in the appropriate area of the `plugin` directory.

The way this drag-and-drop system is achieved will be briefly explained. Within the `plugin` directory, there are three subdirectories: `tools`, `models`, and `integrators`. Within these subdirectories, let's consider the `models` directory as an example. Whenever a new model is added (which is a subdirectory within the `models` subdirectory), the CMake build process will automatically detect it and read the `CMakeLists.txt` file present in the newly added model's subdirectory.

An illustration of the directories is shown below:

```
plugin/
  models/
    model1/
        CMakeLists.txt
    model2/
        CMakeLists.txt
  tools/
  integrators/
```

CMake will then follow the instructions provided by the specified model. Therefore, to delete a model, such as `model1`, simply removing the `model1` subdirectory will remove it, as CMake will no longer be able to read the deleted `CMakeLists.txt` file.

6.2 Fast-Running Codes

When developing a plugin for X, it may involve fast-running code written in another language, such as Fortran, paired with Python to speed up processing time. In this subsection, key functions embedded in the CMake build system will be discussed to help compile these fast-running codes.

6.3 Integrators

Adding an integrator is a straightforward process. The key requirement is to set the cache variable to indicate that the integrator exists. This variable allows models to specify which integrator they require. At build time, CMake checks if the integrator exists; if not, an error is thrown. Additionally, the integrator and any files you wish to install during the installation process must be specified.

For example, let's say you add an integrator, `integrator1`, into the `integrators` directory:

```
plugin/
  models/
  tools/
  integrators/
    integrator1/
        CMakeLists.txt
        integrator.py
```

Within the `CMakeLists.txt` file, you would simply include these lines:

```
1  set(INTEGRATOR1 TRUE)
2  install(FILES integrator.py)
```

6.4 Models

Adding a model is a similar process to adding an integrator. An optional, but recommended, step is to specify any required integrators for the model. This ensures that the build process halts if the necessary integrators are not present. This is achieved by calling the function `need_integrator`. Make sure to specify the installation of any needed files.

In this example, the model `model1` is being added to the `models` directory:

```
plugin/
  models/
    model1/
        CMakeLists.txt
        model1.py
        model1.f90
        model_lib.f90
```

```
tools/
integrators/
  integrator1/
      CMakeLists.txt
      integrator.py
```

Since `model1` requires `integrator1` and is a fast-running Fortran code, we specify the integrator requirement and use the `compile_f2py` function to create the shared object file from the Fortran (`.f90`) source files:

```
1  needs_integrator("integrator1") # for more integrators, separate
       with a space
2
3  set(model_source_file model1.f90)
4  set(model_library_file model_lib.f90)
5
6  compile_f2py(model_source_file model_library_file)
7
8  install(FILES model1.py)
```

When CMake picks up this `model1` CMake file, it will check for the required integrators, then use `f2py` to create the shared object file. This file will be installed (as done within the `compile_f2py` function), followed by the installation of the `model1.py` file.

6.5 Tools

The process of adding a tool is identical to adding a model; however, the `needs_integrator` line is not required.

6.6 Unit Testing

To add unit tests for your desired plugins, create a new subdirectory under `X/test/plugin` with the prefix `test` followed by the name of your unit test. Within this directory, include all unit test source files as well as a `CMakeLists.txt` file, as shown below:

```
test/
  tools/
  models/
  plugin/
    test_plugin1/
      CMakeLists.txt
      test_one.py
      test_two.py
```

It is recommended to create your unit tests using `pytest`, as this is the testing framework used by all other unit tests within X.

Within the `CMakeLists.txt` file, you must include the name of your test.

7 Usage

7.1 Setup

Once X has been successfully built, installed, and the unit tests are passing, it is ready to be used. However, before using it, ensure that the module files generated during the build process are loaded. This ensures that X can be used anywhere on the machine. Always load the `X` module file, and the `Dakota` module file when Dakota is needed.

Note: The Spack environment must still be loaded when using X for its intended purpose.

7.2 Standalone Usage

When using X independently of Dakota, you must call it directly using the following command:

```
1    python3 -m X
```

The output of this command will display the usage information for X, including various flags that can be passed to it.

X operates in two modes: `t` and `k`. The `t` mode must be executed before `k`, as it generates an SQL file that `k` relies on. The `t` mode requires both a JSON input file and the name of the output SQL file.

To run X in t mode:

```
1    python3 -m X t
```

7.3 Input Decks

7.4 Dakota Usage

8 Example Test Cases

9 Appendix

9.1 Additional Resources

- Official Documentation
- Community Forum