LA-UR-22-20363

Approved for public release; distribution is unlimited.

Title:	Spiner
Author(s):	Miller, Jonah Maxwell
Intended for:	Online documentation for open-source software Web
Issued:	2022-01-14

Spiner

The Spiner Team

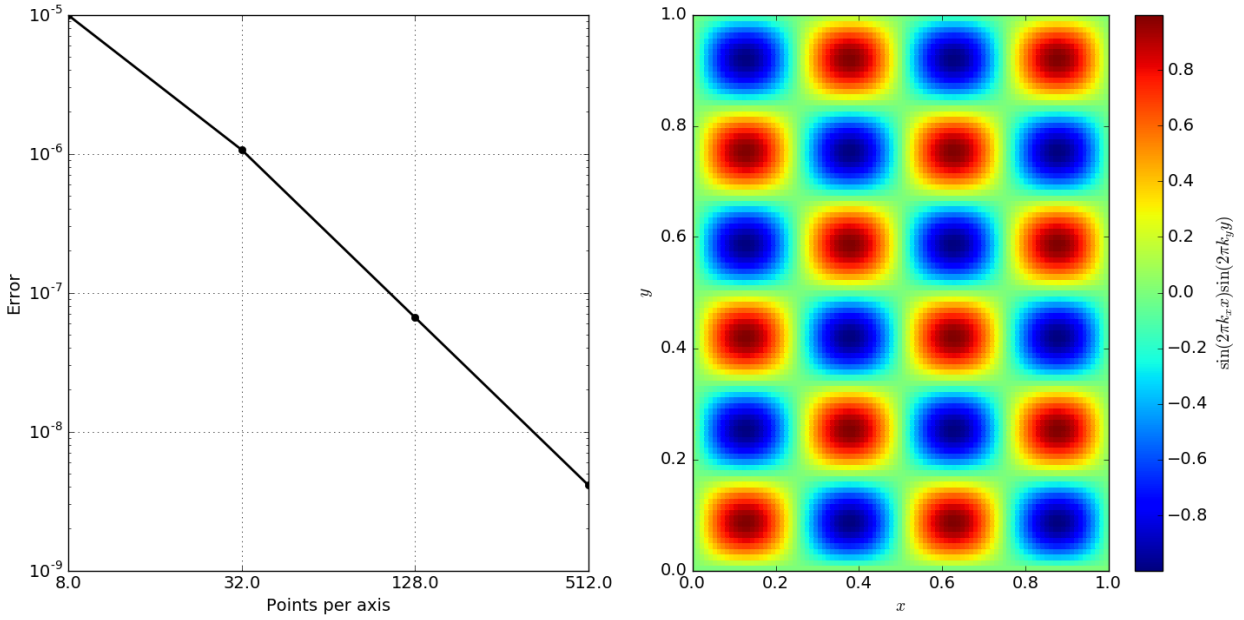
Jan 12, 2022

CONTENTS:

Spiner is a library for storing, indexing, and interpolating multidimensional data in a performance-portable way. It's intended to run on CPUs, GPUs and everything in-between. You can create a table on a CPU, copy it to a GPU, and interpolate on it in a GPU kernel, for example.

Spiner also defines (via hdf5) a file format that bundles data together with instructions for interpolating it. This means you don't have to specify anything to start interpolating, simple load the file and evaluate where you want.

Interpolation is linear. Here's an example of 3D interpolation (2D slice shown) on a GPU, with second-order convergence:



See below for details of how to use spiner in your project and how to develop for it.

CHAPTER

ONE

BUILDING AND INSTALLATION

`Spiner` is self-contained and header-only. Clone it as:

```
git clone git@github.com:lanl/spiner.git
```

1.1 Building from source

To build tests and install:

```
mkdir -p spiner/bin
cd sppiner/bin
cmake -DBUILD_TESTING=ON
make -j
make test
make install
```

Spiner supports a few `cmake` configuration options:

- `BUILD_TESTING` enables tests
- `SPINER_USE_HDF5` enables support for saving and loading tables as hdf5.
- `SPINER_USE_KOKKOS` enables Kokkos as a backend
- `SPINER_USE_CUDA` enables the Kokkos cuda backend
- `CMAKE_INSTALL_PREFIX` sets the install location
- `CMAKE_BUILD_TYPE` sets the build type

HDF5 is searched for and configured via the usual cmake machinery.

A `format` target is also added if `clang-format` is found, so that `make format` will auto-format the repository.

Testing is enabled via Catch2, which is automatically downloaded during the cmake configure phase if needed.

1.2 Spack

Warning: The spack build is currently experimental. Please report problems you have as github issues.

Although the spackage has not yet made it to the main Spack repositories, we provide a spackage for `Spiner` within the the source repository. If you have spack installed, simply call

```
spack repo add spiner/spack-repo
spack install spiner
```

The spack repo supports a few variants:

- `+kokkos` enables the Kokkos backend
- `+cuda` enables the cuda backend. A `cuda_arch` must be specified.
- `+python` installs python, numpy, and matplotlib support
- `+doc` adds tooling for building the docs
- `+format` adds support for clang-format

1.3 Including Spiner in your Project

Spiner can be included into a cmake project, either in-tree as a submodule or after installation. The cmake system provides `spiner::flags` and `spiner::libs` cmake targets. The former adds appropriate compilation flags, the latter adds link flags for dependencies such as hdf5.

CHAPTER

TWO

GETTING STARTED

The following provides a simple example of utilizing a DataBox.

```
#include <iostream>
#include <databox.hpp>
using namespace Spiner;

int main() {
  // create a databox
  constexpr int NX1 = 2;
  constexpr int NX2 = 3;
  constexpr int NX3 = 4;
  DataBox db(NX3, NX2, NX1);

  // fill the databox with the flat index of each element
  for (int i = 0; i < db.size(); ++i) {
    db(i) = static_cast<double>(i);
  }

  // set the interpolation ranges to [0,1] or each dimension
  for (int d = 0; d < db.rank(); ++d) {
    db.setRange(d, 0, 1, db.dim(d));
  }

  // interpolate
  double val = db.interpToReal(0.2, 0.3, 0.4);

  // save to file
  db.saveHDF("my_data.sp5");

  // load a new databox from file
  DataBox db2;
  db2.loadHDF("my_data.sp5");

  // interpolate new databox to the same location
  double val2 = db2.itnerpToReal(0.2, 0.3, 0.4);

  // print the interpolated values and see they're the same
  std::cout << val1 << ", " val2 << ": " << (val1 - val2) << std::endl;

  // free the databoxes
```

(continues on next page)

(continued from previous page)

```
  free(db);
  free(db2);

  return 0;
}
```

For more examples, please consult the test directory.

CHAPTER

THREE

THE DATABOX

The fundamental data type in `spiner` is the `DataBox`. A `DataBox` packages a multi-dimensional (up to six dimensions) array with routines for interpolating on the array and for saving the data to and loading the data from file.

To use databox, simply include the relevant header:

```
#include <databox.hpp>
```

Note: In the function signatures below, GPU/performance portability decorators have been excluded for brevity. However they are present in the actual code.

3.1 Creating a `DataBox`

You can create a `DataBox` of a given shape via the constructor:

```
int nx1 = 2;
int nx2 = 3;
int nx3 = 4;
Spiner::DataBox db(nx3, nx2, nx1);
```

The constructor takes any number of shape values (e.g., `nx*`) up to six (or `Spiner::MAXRANK`) values. Zero shape values initializes an empty, size-zero array.

Note: `DataBox` is column-major ordered. So `x3` is the slowest moving index and `x1` is the fastest.

Note: The data in `DataBox` is always real-valued. It is usually of type `double` but can be set to type `float` if the preprocessor macro `SINGLE_PRECISION_ENABLED` is defined. There is a `Real` typedef that has the same type as the `DataBox` data type.

If GPU support is enabled, a `DataBox` can be allocated on either host or device, depending on the `AllocationTarget`. For example, to explicitly allocate one array on the host and one on the device, you might call:

```
// Allocates on the host (CPU)
Spiner::DataBox db_host(Spiner::AllocationTarget::Host, nx2, nx1);
// Allocates on the device (GPU)
Spiner::DataBox db_dev(Spiner::AllocationTarget::Device, nx2, nx1);
```

Note: If GPU support is not enabled, these both allocate on host.

You can also wrap a `DataBox` around a pointer you allocated yourself. For example:

```
std::vector<double> mydata(nx1*nx2);
Spiner::DataBox db(mydata.data(), nx2, nx1);
```

You can also resize a `DataBox`, which you can use to modify a `DataBox` in-place. For example:

```
Spiner::DataBox db; // empty
// clears old memory, resizes the underlying array,
// and resets strides
db.resize(nx3, nx2, nx1);
```

Just like the constructor, `resize` takes an optional (first) argument for the `AllocationTarget`.

Warning: `DataBox::resize` is destructive. The underlying data is not preserved.

If you want to change the stride without changing the underlying data, you can use `reshape`, which modifies the dimensions of the array, without modifying the underlying memory. For example:

```
// allocate a 1D databox
Spiner::DataBox db(nx3*nx2*nx1);
// interpret it as a 3D object
db.reshape(nx3, nx2, nx1);
```

Warning: Make sure not to change the underlying size of the array when using `reshape`. This is checked with an `assert` statement, so you will get errors when compiling without the `NDEBUG` preprocessor macro.

The method

void `DataBox::reset`();

sets the `DataBox` to be empty with zero rank.

3.2 Copying a `DataBox` to device

If GPU support is enabled, you can deep-copy a `DataBox` and any data contained in it from host to device with the function

DataBox `getOnDeviceDataBox`(DataBox &db_host);

which returns a new databox with the data in `db_host` copied to GPU. An object-oriented method

DataBox `Databox::getOnDevice`() const;

exists as well, which returns a new object with the underlying data copied to GPU.

Note: If GPU support is not enabled, `getOnDevice` and friends are no-ops.

3.3 Semantics and Memory Management

`DataBox` has reference semantics—meaning that copying a `DataBox` does not copy the underlying data. In other words,

```
Spiner::DataBox db1(size);
Spiner::DataBox db2 = db1;
```

shallow-copies `db1` into `db2`. Especially for Kokkos like workflows, this is very useful.

Warning: `DataBox` is neither reference-counted nor garbage-collected. If you create a `DataBox` you must clear the memory allocated just like you would for a pointer.

Two functions are provided for freeing memory in `DataBox`:

void **free**(DataBox &db);

and

`DataBox::`**finalize**();

both will do the same thing and free the memory in a `DataBox` in a context-dependent way. I.e., no matter what the `AllocationTarget` was, the appropriate memory will be freed.

Warning: Do not free a `DataBox` if its memory is managed externally, e.g., via a `std::vector`. `DataBox` checks for this use-case via an `assert` statement.

You can check whether a given `DataBox` is empty, unmanaged, or allocated on host or device with the

`DataBox::`**dataStatus**() const;

method. It returns an `enum class`, `Spiner::DataStatus`, which can take on the values `Empty`, `Unmanaged`, `AllocatedHost`, or `AllocatedDevice`. You can also check whether or not `free` should be called with the method

bool `DataBox::`**ownsAllocatedMemory**();

which returns `true` if a given databox is managing memory and `false` otherwise. The method

bool `DataBox::`**isReference**();

returns `false` if the databox is managing memory and `true` otherwise.

3.4 Using DataBox with smart pointers

Smart pointers can be used to managed a DataBox and automatically call free for you, so long as you use them with a custom deleter. Spiner provides the following deleter for use in this scenario:

```
struct DBDeleter {
  template <typename T>
  void operator()(T *ptr) {
    ptr->finalize();
    delete ptr;
  }
};
```

It can be used, for example, with a std::unique_ptr via:

```
// needed for smart pointers
#include <memory>

// Creates a unique pointer pointing to a DataBox
// with memory allocated on device
std::unique_ptr<DataBox, Spiner::DBDeleter> pdb(
  new DataBox(Spiner::AllocationTarget::Device, N));

// Before using the databox in, e.g., a GPU or Kokkos kernel, get a
// shallow copy:
auto db = *pdb;
// some kokkos code...

// when you leave scope, the data box will be freed.
```

3.5 Accessing Elements of a DataBox

Elements of a DataBox can be accessed and set via the () operator. For example:

```
Spiner::DataBox db(nx3, nx2, nx1);
db(2,1,0) = 5.0;
```

The () operator accepts between one and six indexes. If you pass in more indexes than the rank of the array, the excess indices are ignored. If you pass in fewer, the unset indices are assumed to be zero. The exception is the one-dimensional operator. You can always stride through the "flattened" array by using the one-dimensional accessor. For example:

```
for (int i = 0; i < nx3*nx2*nx1; ++i) {
  db(i) = static_cast<double>(i);
}
```

fills the three-dimensional array above with the flat index of each element.

3.6 Slicing

A new `DataBox` containing a shallow slice of another `DataBox` can be constructed with the `slice` method:

DataBox `DataBox::slice(const int dim, const int indx, const int nvar) const;`

this is fairly limited functionality. It returns a new `DataBox` containing only elements from `indx` to `indx + nvar - 1` in the `dim` direction. All other directions are unchanged. The slowest moving dimension can be sliced to a single index with

DataBox `DataBox::slice(const int indx) const;`

and the slowst-moving two dimensions can be sliced to a single pair of indicies with

DataBox `DataBox::slice(const int i2, int i1) const;`

3.7 Index Types and Interpolation Ranges

Often-times an array mixes "continuous" and "discrete" variables. In other words, some indices of an array are discretizations of a continuous quantity, and we want to interpolate in those directions, but other indices are discrete—they may index a particle species, for example. A common example is in neutrino transport, where an array of emissivities may depend on fluid density, fluid temperature, electron fraction, neutrino energy, and neutrino species. The species can only take three discrete values, but the density, temperature, and electron fraction are all continuous.

`Spiner` accounts for this by assigning each dimension in the array a "type," represented as an `enum class`, `IndexType`. Currently the type can be either `Interpolated` or `Indexed`. When a new `DataBox` is created, all dimensions are set to `IndexType::Indexed`. A dimension can be set to `Interpolated` via the `setRange` method:

void `DataBox::setRange(int i, Real min, Real max, int N) const;`

where here `i` is the dimension, `min` is the minimum value of the independent variable, `max` is the maximum value of the indpendent variable, and `N` is the number of points in the `i` dimension. (Recall that `Real` is usually a typedef to `double`.)

Note: In these routines, the dimension is indexed from zero.

This information can be recovered via the `range` getter method:

void `DataBox::range(int i, Real &min, Real &max, Real &dx, int &N) const;`

where here `min`, `max`, `dx`, and `N` are filled with the values for a given dimension.

Note: There is a lower-level object, `RegularGrid1D`, which represents these interpolation ranges internally. There are setter and getter methods `setRange` and `range` that work with the `RegularGrid1D` class directly. For more details, see the relevant documentation.

It's often desirable to have multiple databoxes with the exact same shape and interpolation structure (i.e., independent variable ranges). In this case, the method

void `DataBox::copyMetadata`(const DataBox &src);

can assist. This method resets and re-allocates the data in a `DataBox` to the exact same size and shape as `src`. More importantly, it also copies the relevant `IndexType` and independent variable range for each dimension.

One can also manually set the `IndexType` in a given dimension with

void `DataBox::setIndexType`(int i, IndexType t);

and retrieve the `IndexType` with

IndexType &`DataBox::indexType`(const int i);

to see if a dimension is interpolatable.

3.8 Interpolation to a real number

The family of `DataBox::interpToReal` methods interpolate the "entire" `DataBox` to a real number. Up to four-dimensional interpolation is supported:

Real `DataBox::interpToReal`(const Real x) const;

Real `DataBox::interpToReal`(const Real x2, const Real x1) const;

Real `DataBox::interpToReal`(const Real x3, const Real x2, const Real x1) const;

Real `DataBox::interpToReal`(const Real x4, const Real x3, const Real x2, const Real x1) const;

where `x1` is the fastest moving direction, `x2` is less fast, and so on. These interpolation routines are hand-tuned for performance.

> **Warning:** Do not call `interpToReal` with a `DataBox` that is the wrong shape or try to interpolate on indices that are not interpolatable. This is checked with an `assert` statement.

3.9 Mixed interpolation and indexing

In the case where an array has some dimensions that are discrete and some that are interpolatable, one can fuse interpolation and indexing into a single operation. These operations are still named `DataBox::interpToReal`, but one of the input arguments is an integer instead of a floating point number. The location of the integer in the function signature indicates which dimension in the `DataBox` is indexed. For example:

Real `DataBox::interpToReal`(const Real x3, const Real x2, const Real x1, const int idx) const;

interpolates the three slower-moving indices and indexes the fastest moving index. On the other hand,

Real `DataBox::interpToReal`(const Real x4, const Real x3, const Real x2, const int idx, const Real x1) const;

interpolates the fastest moving index, then indexes the second-fastest, then interpolates the remaining three slower. The above fused operations are the only ones currently supported.

3.10 Interpolating into another DataBox

There is limited functionality for filling a DataBox with the interpolated values of another DataBox. For example, the method

void DataBox::**interpFromDB**(const DataBox &src, const Real x);

allocates the DataBox to have a rank one lower than src and fill it with the faster moving elements of src interpolated to x in the slowest-moving direction. Similarly for

void DataBox::**interpFromDB**(const DataBox &src, const Real x2, const Real x1);

The methods

DataBox Databox::**InterpToDB**(const Real x) const;

and

DataBox Databox::**InterpToDB**(const Real x2, const Real x1);

return a new DataBox object, rather than setting it from a source DataBox.

3.11 File I/O

If hdf5 is enabled, Spiner can save an array to or load an array from disk. Each array so-saved is also saved with the IndexType and independent variable ranges bundled with it, so that knowledge of how to interpolate the data is automatically available.

The following methods are supported:

herr_t DataBox::**saveHDF**(const std::string &filename) const;

saves the DataBox to a file with filename.

herr_t DataBox::**saveHDF**(hid_t loc, const std::string &groupname) const;

saves the DataBox as an hdf5 group at the location loc in an hdf5 file.

DataBox::**loadHDF**(const std::string &filename);

fills the DataBox from information in the root of a file with filename.

DataBox::**loadHDF**(hid_t loc, const std::string &groupname);

fills the DataBox from information in the group with groupname based at location loc in the file.

3.12 Miscellany

Here we list a few convenience functions available that were not covered elsewhere.

Real `DataBox::min()` const;

and

Real `DataBox::max()` const;

compute and return the minimum and maximum values (respectively) in the array.

int `rank()` const;

returns the rank (number of dimensions) of the array.

int `size()` const;

returns the total number of elements in the underlying array.

int `sizeBytes()` const;

returns the total size of the underlying array in bytes.

int `dim(int i)` const;

returns the size in a given dimension/direction, indexed from zero.

CHAPTER

FOUR

GRIDDING FOR INTERPOLATION

Spiner performs interpolation on uniform, Cartesian-product grids. There is a lower-level object, `RegularGrid1D` which contains the metadata required for these operations. `RegularGrid1D` has a few useful userspace functions, which are described here.

4.1 Construction

A `RegularGrid1D` requires three values to specify an interpolation grid: the minimum value of the independent variable, the maximum value of the independent variable, and the number of points on the grid. These are passed into the constructor:

RegularGrid1D`::`**`RegularGrid1D`**`(`Real min, Real max, size_t N`)`;

Default constructors and copy constructors are also provided.

4.2 Mapping an index to a real number and vice-versa

The function

Real `RegularGrid1D::`**`x`**`(`const int i`)` const;

returns a "physical" position on the grid given an index `i`.

The function

int **`index`**`(`const Real x`)` const;

returns the index on the grid of a "physical" value `x`.

The function

Real **`min`**`()` const;

returns the minimum value on the independent variable grid.

The function

Real **`max`**`()` const;

returns the maximum value on the independent variable grid.

The function

Real **dx**() const;

returns the grid spacing for the independent variable.

The function

Real **nPoints**() const;

returns the number of points in the independent variable grid.

4.3 Developer functionality

For developers, additional functionality is available. Please consult the code.

CHAPTER

FIVE

PORTS OF CALL

Ports of call is a header-only library that provides a bit of flexibility for performance portability. At the moment it mainly provides a one-header abstraction to enable or disable Kokkos in a code. However other backends can be added. (If you're interested in adding a backend, please let us know!)

We define a few portability macros which are useful:

1. `PORTABLE_FUNCTION`: decorators necessary for compiling a kernel function
2. `PORTABLE_INLINE_FUNCTION`: ditto, but for when functions ought to be inlined
3. `PORTABLE_FORCEINLINE_FUNCTION`: forces the compiler to inline
4. `PORTABLE_LAMBDA`: Resolves to a `KOKKOS_LAMBDA` or to `[=]` depending on context
5. `_WITH_KOKKOS_`: Defined if Kokkos is enabled.
6. `_WITH_CUDA_`: Defined when Cuda is enabled
7. `Real`: a typedef to double (default) or float (if you define `SINGLE_PRECISION_ENABLED`)
8. `PORTABLE_MALLOC()`, `PORTABLE_FREE()`: A macro or wrapper for kokkos_malloc or cudaMalloc, or raw malloc.

At compile time, you define `PORTABILITY_STRATEGY_{KOKKOS,CUDA,NONE}` (if you don't define it, it defaults to NONE). The above macros then behave as expected. In particular, `PORTABLE_FUNCTION` and friends resolve to `__host__ __device__` decorators as appropriate.

There are to be two headers in this library, for different use cases.

5.1 portability.hpp

`portability.hpp` provides the above-mentioned macros for decorating functions. Also provides loop abstractions that can be leveraged by a code. These loop abstractions are of the form:

void **`portableFor`**(const char *name, int start, int stop, *Function* Function)

where `Function` is a template parameter and should be set to a functor that takes one index, e.g., an index in an array. For example:

```
portableFor("Example", 0, 5,
  PORTABLE_LAMBDA(int i) {
    printf("hello from thread %d\n", i);
});
```

`start` is inclusive, `stop` is exclusive. Up to five-dimensional `portableFor` loops are available. For example:

```
template <typename Function>
void portableFor(const char *name, int startb, int stopb, int starta, int stopa,
  int startz, int stopz, int starty, int stopy, int startx,
  int stopx, Function function) {
```

We also provide `portableReduce`, however the functionality is very limited. The syntax is:

```
template <typename Function, typename T>
void portableReduce(const char *name, int starta, int stopa, int startz,
  int stopz, int starty, int stopy, int startx, int stopx,
  Function function, T &reduced) {
```

where `Function` now takes as many indices are required and `reduced` as arguments.

5.2 portable_arrays.hpp

`portable_arrays.hpp` provides a wrapper class, `PortableMDArray`, around a contiguous block of host or device memory that knows stride and layout, enabling one to mock up multidimensional arrays from a pointer to memory. The design is heavily inspired by the `AthenaArray` class from Athena++.

One constructs a `PortableMDArray` by passing it a pointer to underlying data and a shape. For example:

```
#include <portability.hpp>
#include <portable_arrays.hpp>
constexpr int NX = 2;
constexpr int NY = 3;
constexpr int NZ = 4;
Real *data = (Real*)PORTABLE_MALLOC(NX*NY*NZ*sizeof(Real));
PortableMDArray<Real> my_3d_array(data, NZ, NY, NX);
```

Note: `PortableMDArray` is templated on underlying data type.

Note:

`PortableMDArray` is column-major-ordered. The

slowest moving index is `z` and the fastest is `x`.

You can then set or access an element by reference as:

```
// z = 3, y = 2, x = 1
my_3d_array(3,2,1) = 5.0;
```

You can always access the "flat" array by simply using the 1D accessor:

```
my_3d_array(6) = 2.0;
```

By default `PortableMDArray` has reference-semantics. In other words, copies are shallow.

You can assign new data and a new shape to a `PortableMDArray` with the `NewPortableMDArray` function. For example:

```
my_3d_array.NewPortableArray(new_data, 9, 8, 7);
```

would reshape `my_3d_array` to be of shape 7x8x9 and point it at the `new_data` pointer.

`PortableMDArray` also provides a few useful methods:

size_t `PortableMDArray::`**`GetRank`**`()`

provides the number of dimensions of the array.

int `PortableMDArray::`**`GetDim`**`(`size_t i`)`

returns the size of a given dimension (indexed from 1, not 0).

int `PortableMDArray::`**`GetSize`**`()`

returns the size of the flattened array.

size_t `PortableMDArray::`**`GetSizeInBytes`**`()`

returns the size of the flattened array in bytes.

bool `PortableMDArray::`**`IsEmpty`**`()`

returns true if the array is empty and false otherwise.

T *`PortableMDArray::`**`data`**`()`

returns the underlying pointer. The `begin()` and `end()` functions return pointers to the beginning and end of the array.

void `PortableMDArray::`**`Reshape`**`(`int nx3, int nx2, int nx1`)`

resets the shape of the array without pointing to a new underlying data pointer. It accepts anywhere between 1 and 6 sizes.

`PortableMDArray` also supports some simple boolean comparitors, such as == and arithmetic such as +, and -.

CHAPTER

SIX

HOW TO USE SPHINX FOR WRITING DOCS

6.1 How to Get the Dependencies

6.1.1 Using Docker

If you are using Docker, then simply pull the docker image specified below:

```
image: sphinxdoc/sphinx-latexpdf
```

Then, after running `docker run -it <docker-image-name> /bin/bash`, install the theme we are using with `pip install sphinx_rtd_theme`

6.1.2 Using Spack

If you are using Spack to provision dependencies, then follow the steps as such:

```
    - cd ${CI_PROJECT_DIR}/doc/sphinx
    - make html
    - rm -rf ${CI_PROJECT_DIR}/public
    - mv _build/html ${CI_PROJECT_DIR}/public

.test:
  stage: build_n_test
  extends:
```

from `.gitlab-ci.yml`

> **Warning:** If you do not have either Docker or Spack locally, you would need to install one of them first.
>
> For Docker, refer to their Get Docker Guide.
>
> For Spack, refer to their Getting Started Guide.

6.1.3 Using Python

With your favorite python package manager, e.g., `pip`, install `sphinx`, `spinx_multiversion`, and `sphinx_rtd_theme`. For example:

```
pip install sphinx
pip install sphinx_multiversion
pip install sphinx_rtd_theme
```

6.2 How to Build .rst into .html

After you have the dependencies in your environment, then simply build your documentation as the following:

```
make html
```

from `.gitlab-ci.yml`

Note: You can view the documentation webpage locally on your web browser by passing in the URL as `file:///path/to/spiner/doc/sphinx/_build/html/index.html`

6.3 How to Deploy

1. Submit a PR with your .rst changes for documentation on Github Spiner
2. Get your PR reviewed and merged into main
3. Make sure the `pages` CI job passes in the CI pipeline

As soon as the PR is merged into main, this will trigger the Pages deployment automatically if the `pages` CI job passes.

Documentation is available on github-pages and on re-git

6.4 More Info.

- Sphinx Installation
- Sphinx reStructuredText Documentation

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

INDEX