NetSci

Generated by Doxygen 1.9.8

1 NetSci: A Toolkit for High Performance Scientific Network Analysis Computation **1**
1.1 Overview . . . 1
1.2 Installation . . . 2
2 Namespace Index **2**
2.1 Namespace List . . . 2
3 Class Index **3**
3.1 Class List . . . 3
4 Namespace Documentation **3**
4.1 netcalc Namespace Reference . . . 3
4.1.1 Detailed Description . . . 4
4.1.2 Function Documentation . . . 4
5 Class Documentation **8**
5.1 Atom Class Reference . . . 8
5.1.1 Constructor & Destructor Documentation . . . 9
5.1.2 Member Function Documentation . . . 10
5.2 Atoms Class Reference . . . 15
5.2.1 Constructor & Destructor Documentation . . . 16
5.2.2 Member Function Documentation . . . 16
5.3 CuArray< T > Class Template Reference . . . 17
5.3.1 Detailed Description . . . 19
5.3.2 Constructor & Destructor Documentation . . . 20
5.3.3 Member Function Documentation . . . 20
5.4 CuArrayRow< T > Class Template Reference . . . 54
5.5 Network Class Reference . . . 55
5.5.1 Constructor & Destructor Documentation . . . 55
5.5.2 Member Function Documentation . . . 56
5.6 Node Class Reference . . . 59
5.6.1 Detailed Description . . . 60
5.6.2 Constructor & Destructor Documentation . . . 60
5.6.3 Member Function Documentation . . . 61
Index **63**

1 NetSci: A Toolkit for High Performance Scientific Network Analysis Computation

1.1 Overview

NetSci is a specialized toolkit designed for advanced network analysis in computational sciences. Utilizing the capabilities of modern GPUs, it offers a powerful and efficient solution for processing computationally demanding network analysis metrics while delivering state-of-the-art performance.

1.2 Installation

NetSci is designed with a focus on ease of installation and long-term stability, ensuring compatibility with Linux systems featuring CUDA-capable GPUs (compute capability 3.5 and above). It leverages well-supported core C++ and Python libraries to maintain simplicity and reliability.

1. **Download Miniconda Installation Script**:
   ```
   wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
   ```
2. **Execute the Installation Script**:
   ```
   bash Miniconda3-latest-Linux-x86_64.sh
   ```
3. **Update Environment Settings**:
   ```
   source ~/.bashrc
   ```
4. **Install Git with Conda**:
   ```
   conda install -c conda-forge git
   ```
5. **Clone the NetSci Repository**:
   ```
   git clone https://github.com/netscianalysis/netsci.git
   ```
6. **Navigate to the NetSci Root Directory**:
   ```
   cd netsci
   ```
7. **Create NetSci Conda Environment**:
   ```
   conda env create -f netsci.yml
   ```
8. **Activate NetSci Conda Environment**:
   ```
   conda activate netsci
   ```
9. **Create CMake Build Directory**:
   ```
   mkdir build
   ```
10. **Set NetSci Root Directory Variable**:
    ```
    NETSCI_ROOT=$(pwd)
    ```
11. **Navigate to the CMake Build Directory**:
    ```
    cd ${NETSCI_ROOT}/build
    ```
12. **Compile CUDA Architecture Script**:
    ```
    nvcc ${NETSCI_ROOT}/build_scripts/cuda_architecture.cu -o cuda_architecture
    ```
13. **Set CUDA Architecture Variable**:
    ```
    CUDA_ARCHITECTURE=$(./cuda_architecture)
    ```
14. **Configure the Build with CMake**:
    ```
    cmake .. -DCONDA_DIR=$CONDA_PREFIX -DCUDA_ARCHITECTURE=${CUDA_ARCHITECTURE}
    ```
15. **Build NetSci**:
    ```
    cmake --build . -j
    ```
16. **Build NetSci Python Interface**:
    ```
    make python
    ```
17. **Test C++ and CUDA Backend**:
    ```
    ctest
    ```
18. **Run Python Interface Tests**:
    ```
    cd ${NETSCI_ROOT}
    pytest
    ```

2 Namespace Index

2.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

netcalc
The netcalc namespace 3

3 Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Atom 8

Atoms 15

CuArray< T >
Manages CUDA-supported arrays, offering initialization, memory management, and data manipulation. Implemented as a template class in C++, with Python and Tcl wrapper interfaces. In Python and Tcl, use as <ElementType>CuArray (e.g., FloatCuArray, IntCuArray), as they don't support templates. Supports float and int types in Python and Tcl, and all numeric types in C++ 17

CuArrayRow< T > 54

Network 55

Node
Represents a node in a graph 59

4 Namespace Documentation

4.1 netcalc Namespace Reference

The netcalc namespace.

Functions

- int mutualInformation (CuArray< float > *X, CuArray< float > *I, CuArray< int > *ab, int k, int n, int xd, int d, int platform)

  Computes the mutual information between all pairs of random variables listed in 'ab'.
- int mutualInformation (CuArray< float > *X, CuArray< float > *I, CuArray< int > *ab, int k, int n, int xd, int d, int platform, int checkpointFrequency, std::string checkpointFileName)

  Computes the mutual information between all pairs of random variables listed in 'ab'.
- float mutualInformationGpu (CuArray< float > *Xa, CuArray< float > *Xb, int k, int n, int xd, int d)

  Computes the mutual information between two random variables Xa and Xb on the GPU.
- float mutualInformationCpu (CuArray< float > *Xa, CuArray< float > *Xb, int k, int n, int xd, int d)

  Computes the mutual information between two random variables Xa and Xb on the CPU.
- void generateRestartAbFromCheckpointFile (CuArray< int > *ab, CuArray< int > *restartAb, const std::string &checkpointFileName)

  Creates an ab array of nodes that still need to have their mutual information/generalized correlation calculated, using a mutualInformation or generalizedCorrelation checkpoint file.
- int generalizedCorrelation (CuArray< float > *X, CuArray< float > *R, CuArray< int > *ab, int k, int n, int xd, int d, int platform, int checkpointFrequency, std::string checkpointFileName)

Computes the generalized correlation between all pairs of random variables listed in 'ab'.

- int **generalizedCorrelation** (CuArray< float > *X, CuArray< float > *R, CuArray< int > *ab, int k, int n, int xd, int d, int platform)
- float generalizedCorrelationGpu (CuArray< float > *Xa, CuArray< float > *Xb, int k, int n, int xd, int d)

  Computes the generalized correlation between two random variables Xa and Xb on the GPU.
- float generalizedCorrelationCpu (CuArray< float > *Xa, CuArray< float > *Xb, int k, int n, int xd, int d)

  Computes the generalized correlation between two random variables Xa and Xb on the CPU.

4.1.1 Detailed Description

The netcalc namespace.

4.1.2 Function Documentation

generalizedCorrelation()

```
int netcalc::generalizedCorrelation (
            CuArray< float > * X,
            CuArray< float > * R,
            CuArray< int > * ab,
            int k,
            int n,
            int xd,
            int d,
            int platform,
            int checkpointFrequency,
            std::string checkpointFileName )
```

Computes the generalized correlation between all pairs of random variables listed in 'ab'.

Parameters

X	Mx(d*N) matrix of M d-dimensional random variables with N samples.
R	Vector that stores the generalized correlation between pairs of random variables listed in 'ab'.
ab	Vector of pairs of random variables for which generalized correlation is computed.
k	K value used in generalized correlation calculation.
n	Number of samples.
xd	The dimension of the joint random variable. Only 2D-joint random variables are supported.
d	The dimension of each random variable. Only 1, 2, and 3-dimensional random variables are supported.
platform	Platform (CPU or GPU) used for computation. Use 0 for GPU, and 1 for CPU.

Returns

0 if successful, 1 otherwise.

generalizedCorrelationCpu()

```
float netcalc::generalizedCorrelationCpu (
            CuArray< float > * Xa,
```

```
            CuArray< float > * Xb,
            int k,
            int n,
            int xd,
            int d )
```

Computes the generalized correlation between two random variables Xa and Xb on the CPU.

Parameters

Xa	CuArray representing the first random variable.
Xb	CuArray representing the second random variable.
k	K value used in generalized correlation calculation.
n	Number of samples.
xd	The dimension of the joint random variable. Only 2D-joint random variables are supported.
d	The dimension of each random variable. Only 1, 2, and 3-dimensional random variables are supported.

Returns

The computed generalized correlation value.

generalizedCorrelationGpu()

```
float netcalc::generalizedCorrelationGpu (
            CuArray< float > * Xa,
            CuArray< float > * Xb,
            int k,
            int n,
            int xd,
            int d )
```

Computes the generalized correlation between two random variables Xa and Xb on the GPU.

Parameters

Xa	CuArray representing the first random variable.
Xb	CuArray representing the second random variable.
k	K value used in generalized correlation calculation.
n	Number of samples.
xd	The dimension of the joint random variable. Only 2D-joint random variables are supported.
d	The dimension of each random variable. Only 1, 2, and 3-dimensional random variables are supported.

Returns

The computed generalized correlation value.

generateRestartAbFromCheckpointFile()

```
void netcalc::generateRestartAbFromCheckpointFile (
            CuArray< int > * ab,
```

```
            CuArray< int > * restartAb,
            const std::string & checkpointFileName )
```

Creates an ab array of nodes that still need to have their mutual information/generalized correlation calculated, using a mutualInformation or generalizedCorrelation checkpoint file.

Parameters

ab	Original ab array.
restartAb	The ab array of nodes that still need to have their mutual information/generalized correlation calculated.
checkpointFileName	The name of the checkpoint file.

mutualInformation() [1/2]

```
int netcalc::mutualInformation (
            CuArray< float > * X,
            CuArray< float > * I,
            CuArray< int > * ab,
            int k,
            int n,
            int xd,
            int d,
            int platform )
```

Computes the mutual information between all pairs of random variables listed in 'ab'.

Parameters

X	Mx(d∗N) matrix of M d-dimensional random variables with N samples.
I	Vector that stores the mutual information between pairs of random variables listed in 'ab'.
ab	Vector of pairs of random variables for which mutual information is computed.
k	K value used in mutual information calculation.
n	Number of samples.
xd	The dimension of the joint random variable. Only 2D-joint random variables are supported.
d	The dimension of each random variable. Only 1, 2, and 3-dimensional random variables are supported.
platform	Platform (CPU or GPU) used for computation. Use 0 for GPU, and 1 for CPU.

Returns

0 if successful, 1 otherwise.

mutualInformation() [2/2]

```
int netcalc::mutualInformation (
            CuArray< float > * X,
            CuArray< float > * I,
            CuArray< int > * ab,
            int k,
```

```
            int n,
            int xd,
            int d,
            int platform,
            int checkpointFrequency,
            std::string checkpointFileName )
```

Computes the mutual information between all pairs of random variables listed in 'ab'.

Parameters

X	Mx(d∗N) matrix of M d-dimensional random variables with N samples.
I	Vector that stores the mutual information between pairs of random variables listed in 'ab'.
ab	Vector of pairs of random variables for which mutual information is computed.
k	K value used in mutual information calculation.
n	Number of samples.
xd	The dimension of the joint random variable. Only 2D-joint random variables are supported.
d	The dimension of each random variable. Only 1, 2, and 3-dimensional random variables are supported.
platform	Platform (CPU or GPU) used for computation. Use 0 for GPU, and 1 for CPU.
checkpointFrequency	Saves the intermediate results after every 'checkpointFrequency' number of iterations.
checkpointFileName	The filename to save the intermediate results. The filename is suffixed with the last ab node pair index the mutual information was calculated for.

Returns

0 if successful, 1 otherwise.

mutualInformationCpu()

```
float netcalc::mutualInformationCpu (
            CuArray< float > * Xa,
            CuArray< float > * Xb,
            int k,
            int n,
            int xd,
            int d )
```

Computes the mutual information between two random variables Xa and Xb on the CPU.

Parameters

Xa	CuArray representing the first random variable.
Xb	CuArray representing the second random variable.
k	K value used in mutual information calculation.
n	Number of samples.
xd	The dimension of the joint random variable. Only 2D-joint random variables are supported.
d	The dimension of each random variable. Only 1, 2, and 3-dimensional random variables are supported.

Returns

The computed mutual information value.

mutualInformationGpu()

```
float netcalc::mutualInformationGpu (
            CuArray< float > * Xa,
            CuArray< float > * Xb,
            int k,
            int n,
            int xd,
            int d )
```

Computes the mutual information between two random variables Xa and Xb on the GPU.

Parameters

Xa	CuArray representing the first random variable.
Xb	CuArray representing the second random variable.
k	K value used in mutual information calculation.
n	Number of samples.
xd	The dimension of the joint random variable. Only 2D-joint random variables are supported.
d	The dimension of each random variable. Only 1, 2, and 3-dimensional random variables are supported.

Returns

The computed mutual information value.

5 Class Documentation

5.1 Atom Class Reference

Public Member Functions

- Atom ()
  Default constructor for Atom.
- Atom (const std::string &pdbLine)
  Constructor for Atom with PDB line.
- Atom (const std::string &pdbLine, int atomIndex)
  Constructor for Atom with PDB line and atom index.
- int index () const
  Get the atom index.
- std::string name ()
  Get the atom name.
- std::string element ()
  Get the atom element.
- std::string residueName ()
  Get the residue name.

- int residueId () const
  Get the residue ID.
- std::string chainId ()
  Get the chain ID.
- std::string segmentId ()
  Get the segment ID.
- float temperatureFactor () const
  Get the temperature factor.
- float occupancy () const
  Get the occupancy.
- int serial () const
  Get the serial number.
- std::string tag ()
  Get the atom tag.
- float mass () const
  Get the mass of the atom.
- unsigned int hash () const
  Get the hash of the atom.
- float x (CuArray< float > *coordinates, int frame, int numFrames) const
  Get the x-coordinate of the atom.
- float y (CuArray< float > *coordinates, int frame, int numFrames) const
  Get the y-coordinate of the atom.
- float z (CuArray< float > *coordinates, int frame, int numFrames) const
  Get the z-coordinate of the atom.
- void load (const std::string &jsonFile)
  Load atom information from a JSON file.

Private Attributes

- int **_index**
- std::string **_name**
- std::string **_element**
- std::string **_residueName**
- int **_residueId**
- std::string **_chainId**
- std::string **_segmentId**
- float **_temperatureFactor**
- float **_occupancy**
- int **_serial**
- std::string **_tag**
- float **_mass**
- unsigned int **_hash**

5.1.1 Constructor & Destructor Documentation

Atom() [1/3]

```
Atom::Atom ( )
```

Default constructor for Atom.

Constructs an empty Atom object.

Atom() [2/3]

```
Atom::Atom (
            const std::string & pdbLine )  [explicit]
```

Constructor for Atom with PDB line.

Constructs an Atom object using the provided PDB line. The constructor parses the PDB line to extract the relevant atom information, such as index, name, element, residue name, residue ID, chain ID, segment ID, temperature factor, occupancy, serial number, atom tag, mass, and hash.

The PDB line should follow the standard PDB format for ATOM records as described in Section 9 of the PDB file format documentation:

See also

https://www.wwpdb.org/documentation/file-format-content/format33/sect9.html#ATOM

Parameters

pdbLine	The PDB line containing atom information in the standard PDB format.

Atom() [3/3]

```
Atom::Atom (
            const std::string & pdbLine,
            int atomIndex )
```

Constructor for Atom with PDB line and atom index.

Constructs an Atom object using the provided PDB line and atom index.

Parameters

pdbLine	The PDB line containing atom information.
atomIndex	The atom index.

5.1.2 Member Function Documentation

chainId()

```
std::string Atom::chainId ( )
```

Get the chain ID.

Returns the chain ID.

Returns

The chain ID.

element()

```
std::string Atom::element ( )
```

Get the atom element.

Returns the atom element.

Returns

The atom element.

hash()

```
unsigned int Atom::hash ( ) const
```

Get the hash of the atom.

Returns the hash of the atom, which is calculated from the atom tag concatenated with the atom index.

Returns

The hash of the atom.

index()

```
int Atom::index ( ) const
```

Get the atom index.

Returns the atom index.

Returns

The atom index.

load()

```
void Atom::load (
            const std::string & jsonFile )
```

Load atom information from a JSON file.

Loads atom information from the specified JSON file.

Parameters

jsonFile	The name of the JSON file to load.

mass()

```
float Atom::mass ( ) const
```

Get the mass of the atom.

Returns the mass of the atom.

Returns

The mass of the atom.

name()

```
std::string Atom::name ( )
```

Get the atom name.

Returns the atom name.

Returns

The atom name.

occupancy()

```
float Atom::occupancy ( ) const
```

Get the occupancy.

Returns the occupancy.

Returns

The occupancy.

residueId()

```
int Atom::residueId ( ) const
```

Get the residue ID.

Returns the residue ID.

Returns

The residue ID.

residueName()

```
std::string Atom::residueName ( )
```

Get the residue name.

Returns the residue name.

Returns

The residue name.

segmentId()

```
std::string Atom::segmentId ( )
```

Get the segment ID.

Returns the segment ID.

Returns

The segment ID.

serial()

```
int Atom::serial ( ) const
```

Get the serial number.

Returns the serial number, which is one greater than the atom index.

Returns

The serial number.

tag()

```
std::string Atom::tag ( )
```

Get the atom tag.

Returns the atom tag, which is the concatenation of the residue name, residue ID, chain ID, and segment ID.

Returns

The atom tag.

temperatureFactor()

```
float Atom::temperatureFactor ( ) const
```

Get the temperature factor.

Returns the temperature factor.

Returns

The temperature factor.

x()

```
float Atom::x (
            CuArray< float > * coordinates,
            int frame,
            int numFrames ) const
```

Get the x-coordinate of the atom.

Returns the x-coordinate of the atom at the specified frame.

Parameters

coordinates	The CuArray containing the coordinates.
frame	The frame index.
numFrames	The total number of frames.

Returns

The x-coordinate of the atom.

y()

```
float Atom::y (
            CuArray< float > * coordinates,
            int frame,
            int numFrames ) const
```

Get the y-coordinate of the atom.

Returns the y-coordinate of the atom at the specified frame.

Parameters

coordinates	The CuArray containing the coordinates.
frame	The frame index.
numFrames	The total number of frames.

Returns

The y-coordinate of the atom.

z()

```
float Atom::z (
            CuArray< float > * coordinates,
            int frame,
            int numFrames ) const
```

Get the z-coordinate of the atom.

Returns the z-coordinate of the atom at the specified frame.

Parameters

coordinates	The CuArray containing the coordinates.
frame	The frame index.
numFrames	The total number of frames.

Returns

The z-coordinate of the atom.

The documentation for this class was generated from the following file:

- atom.h

5.2 Atoms Class Reference

Public Member Functions

- Atoms ()
  Default constructor for Atoms.
- void addAtom (Atom *atom)
  Add an Atom to the Atoms collection.
- int numAtoms () const
  Get the number of Atoms in the collection.
- Atom * at (int atomIndex)
  Get the Atom with the specified index.
- int numUniqueTags () const
  Get the number of unique Atom tags.
- std::vector< Atom * > & atoms ()
  Get a reference to the vector of Atoms.

Private Attributes

- std::vector< Atom * > **atoms_**
- std::set< std::string > **uniqueTags_**

5.2.1 Constructor & Destructor Documentation

Atoms()

```
Atoms::Atoms ( )
```

Default constructor for Atoms.

Constructs an empty Atoms object.

5.2.2 Member Function Documentation

addAtom()

```
void Atoms::addAtom (
            Atom * atom )
```

Add an Atom to the Atoms collection.

Adds the specified Atom to the collection of Atoms.

Parameters

atom	Pointer to the Atom to add.

at()

```
Atom * Atoms::at (
            int atomIndex )
```

Get the Atom with the specified index.

Returns a pointer to the Atom with the specified index.

Parameters

atomIndex	The index of the Atom.

Returns

A pointer to the Atom with the specified index.

atoms()

```
std::vector< Atom * > & Atoms::atoms ( )
```

Get a reference to the vector of Atoms.

Returns a reference to the vector of Atoms.

Returns

A reference to the vector of Atoms.

numAtoms()

```
int Atoms::numAtoms ( ) const
```

Get the number of Atoms in the collection.

Returns the number of Atoms in the collection.

Returns

The number of Atoms.

numUniqueTags()

```
int Atoms::numUniqueTags ( ) const
```

Get the number of unique Atom tags.

Returns the number of unique Atom tags. Atoms with the same tag belong to the same Node.

Returns

The number of unique Atom tags.

The documentation for this class was generated from the following file:

- atoms.h

5.3 CuArray< T > Class Template Reference

Manages CUDA-supported arrays, offering initialization, memory management, and data manipulation. Implemented as a template class in C++, with Python and Tcl wrapper interfaces. In Python and Tcl, use as <↩ ElementType>CuArray (e.g., FloatCuArray, IntCuArray), as they don't support templates. Supports float and int types in Python and Tcl, and all numeric types in C++.

```
#include <cuarray.h>
```

Public Member Functions

- CuArray ()

  Constructs an empty CuArray object.
- CuArrayError init (int m, int n)

  Initializes CuArray with specified dimensions and allocates memory on host and device.
- CuArrayError init (T *host, int m, int n)

  Initializes CuArray with specified host data and dimensions, performing a shallow copy. Allocates memory on both the host and the device. The data is shallow copied, so the ownership remains unchanged.
- CuArrayError fromCuArrayShallowCopy (CuArray< T > *cuArray, int start, int end, int m, int n)

  Performs a shallow copy of data from another CuArray within a specified row range. Copies the host data from the given CuArray, within the inclusive range specified by 'start' and 'end'. This CuArray does not own the copied data, and deallocation is handled by the source CuArray.
- CuArrayError fromCuArrayDeepCopy (CuArray< T > *cuArray, int start, int end, int m, int n)

  Performs a deep copy of data from another CuArray within a specified row range. Copies the host data from the given CuArray, including all data within the inclusive range defined by 'start' and 'end'. Memory for the copied data is allocated in this CuArray's host memory.
- ~**CuArray** ()

  Destructor for CuArray. Deallocates memory on both the host and the device.
- int n () const

  Returns the number of columns in the CuArray.
- int m () const

  Returns the number of rows in the CuArray.
- int size () const

  Returns the total number of elements in the CuArray.
- size_t bytes () const

  Returns the total size in bytes of the CuArray data.
- T *& host ()

  Returns a reference to the host data.
- T *& device ()

  Returns a reference to the device data.
- CuArrayError allocateHost ()

  Allocates memory for the host data.
- CuArrayError allocateDevice ()

  Allocates memory for the device data.
- CuArrayError allocatedHost () const

  Checks if memory is allocated for the host data.
- CuArrayError allocatedDevice () const

  Checks if memory is allocated for the device data.
- CuArrayError toDevice ()

  Copies data from the host to the device.
- CuArrayError toHost ()

  Copies data from the device to the host.
- CuArrayError deallocateHost ()

  Deallocates memory for the host data.
- CuArrayError deallocateDevice ()

  Deallocates memory for the device data.
- CuArrayError fromNumpy (T *NUMPY_ARRAY, int NUMPY_ARRAY_DIM1, int NUMPY_ARRAY_DIM2)

  Copies data from a NumPy array to the CuArray.
- CuArrayError fromNumpy (T *NUMPY_ARRAY, int NUMPY_ARRAY_DIM1)

  Copies data from a NumPy array to the CuArray.
- void toNumpy (T **NUMPY_ARRAY, int **NUMPY_ARRAY_DIM1, int **NUMPY_ARRAY_DIM2)

Copies data from the CuArray to a NumPy array.

- void toNumpy (T **NUMPY_ARRAY, int **NUMPY_ARRAY_DIM1)

  Copies data from the CuArray to a NumPy array.
- T get (int i, int j) const

  Returns the value at a specified position in the CuArray.
- CuArrayError set (T value, int i, int j)

  Sets a value at a specified position in the CuArray.
- CuArrayError load (const std::string &fname)

  Loads CuArray data from a specified file.
- void save (const std::string &fname)

  Saves CuArray data to a specified file.
- CuArray< T > * sort (int i)

  Sorts CuArray based on the values in a specified row.
- T & operator[ ] (int i) const

  Returns a reference to the element at a specified index in the CuArray.
- int owner () const

  Returns the owner status of the CuArray. Indicates whether the CuArray is responsible for memory deallocation.
- CuArray< int > * argsort (int i)

  Performs an argsort on a specified row of the CuArray. Returns a new CuArray containing sorted indices.

Private Attributes

- T * **host_**
- T * **device_**
- int **n_** {}
- int **m_** {}
- int **size_** {}
- size_t **bytes_** {}
- int **allocatedDevice_** {}
- int **allocatedHost_** {}
- int **owner_** {}

5.3.1 Detailed Description

template<typename T>
class CuArray< T >

Manages CUDA-supported arrays, offering initialization, memory management, and data manipulation. Implemented as a template class in C++, with Python and Tcl wrapper interfaces. In Python and Tcl, use as <↩ ElementType>CuArray (e.g., FloatCuArray, IntCuArray), as they don't support templates. Supports float and int types in Python and Tcl, and all numeric types in C++.

Parameters

T	Data type of the array elements.

5.3.2 Constructor & Destructor Documentation

CuArray()

```
template<typename T >
CuArray< T >::CuArray ( )
```

Constructs an empty CuArray object.

C++ Example

```
#include <iostream>
#include "cuarray.h"

int main() {
    std::cout
            « "Running "
            « __FILE__
            « std::endl;

    /* Creates a new float CuArray instance */
    auto *cuArray = new CuArray<float>();

    /* Free memory */
    delete cuArray;

    return 0;
}
```

Python Example

```
"""
Always precede CuArray with the data type
Here we are importing CuArray int and float templates.
"""
from cuarray import FloatCuArray, IntCuArray

print("Running", __file__)

"""Create a new float CuArray instance"""
float_cuarray = FloatCuArray()

"""Create a new int CuArray instance"""
int_cuarray = IntCuArray()
```

5.3.3 Member Function Documentation

allocatedDevice()

```
template<typename T >
CuArrayError CuArray< T >::allocatedDevice ( ) const
```

Checks if memory is allocated for the device data.

Returns

CuArrayError indicating success or failure of the operation.

C++ Example

```
#include <cuarray.h>
#include <iostream>
#include <random>

int main() {

    std::cout
```

```
            « "Running "
            « __FILE__
            « std::endl;

/* Creates a new float CuArray instance */
    CuArray<float> *cuArray = new CuArray<float>();

/* Initialize the CuArray with 300 rows and 300 columns */
    auto rows = 300;
    auto cols = 300;
    cuArray->init(rows,
                  cols);

/* Allocate device memory. */
    cuArray->allocateDevice();

/* Check if device memory is allocated. If it is,
 * allocatedDevice() will return 1, other wise it
 * will return 0. This is convenient for boolean checks.*/
    auto deviceMemoryAllocated = cuArray->allocatedDevice();

/* Print whether or not device memory is allocated. */
    std::cout
            « "Device memory allocated: "
            « deviceMemoryAllocated
            « std::endl;

    delete cuArray;

    return 0;
}
```

allocateDevice()

```
template<typename T >
CuArrayError CuArray< T >::allocateDevice ( )
```

Allocates memory for the device data.

Returns

CuArrayError indicating success or failure of the operation.

C++ Example

```
#include <cuarray.h>
#include <random>
#include <iostream>

int main() {
    std::cout
            « "Running "
            « __FILE__
            « std::endl;
/* Creates a new float CuArray instance */
    CuArray<float> *cuArray = new CuArray<float>();

/* Initialize the CuArray with 300 rows and 300 columns */
    auto rows = 300;
    auto cols = 300;
    cuArray->init(rows,
                  cols);

/* Fill the CuArray with random values */
    for (int i = 0; i < cuArray->m(); i++) {
        for (int j = 0; j < cuArray->n(); j++) {
            cuArray->host()[i * cuArray->n() + j] =
                    static_cast<float>(rand() / (float) RAND_MAX);
        }
    }
/* Allocate device memory. If successful, allocateDevice returns 0.*/
    auto err = cuArray->allocateDevice();

    /* Check if device memory allocation was successful. */
    if (err == 0) {
        std::cout
```

```
                « "Device memory allocated successfully."
                « std::endl;
    } else {
        std::cout
                « "Device memory allocation failed."
                « std::endl;
    }

/* Frees host and device memory. */
    delete cuArray;
    return 0;
}
```

allocatedHost()

```
template<typename T >
CuArrayError CuArray< T >::allocatedHost ( ) const
```

Checks if memory is allocated for the host data.

Returns

CuArrayError indicating success or failure of the operation.

C++ Example

```
#include <cuarray.h>
#include <random>
#include <iostream>

int main() {
    std::cout
            « "Running "
            « __FILE__
            « std::endl;
/* Creates a new float CuArray instance */
    CuArray<float> *cuArray = new CuArray<float>();

/* Initialize the CuArray with 300 rows and 300 columns */
    auto rows = 300;
    auto cols = 300;
    cuArray->init(rows,
                  cols);

/* Check if host memory is allocated. If it is,
 * allocatedHost() will return 1, other wise it
 * will return 0. This is convenient for boolean checks.*/
    auto hostMemoryAllocated = cuArray->allocatedHost();

/* Print whether or not host memory is allocated. */
    std::cout
            « "Host memory allocated: "
            « hostMemoryAllocated
            « std::endl;

    delete cuArray;
    return 0;
}
```

allocateHost()

```
template<typename T >
CuArrayError CuArray< T >::allocateHost ( )
```

Allocates memory for the host data.

Returns

CuArrayError indicating success or failure of the operation.

C++ Example

```
#include <cuarray.h>
#include <random>
#include <iostream>

int main() {
    std::cout
            « "Running "
            « __FILE__
            « std::endl;

/* Creates a new float CuArray instance */
    CuArray<float> *cuArray = new CuArray<float>();

/* Initialize the CuArray with 300 rows and 300 columns */
    auto rows = 300;
    auto cols = 300;
    cuArray->init(rows,
                  cols);

/* Fill the CuArray with random values */
    for (int i = 0; i < cuArray->m(); i++) {
        for (int j = 0; j < cuArray->n(); j++) {
            cuArray->host()[i * cuArray->n() + j] =
                    static_cast<float>(rand() / (float) RAND_MAX);
        }
    }
/* Allocate device memory. */
    cuArray->allocateDevice();

/* Copy data from host to device. */
    cuArray->toDevice();

/* Free host memory, since it is no longer needed.*/
    cuArray->deallocateHost();

/*Do some complicated GPU calculations
* and then allocate host memory when you need it again.
* Also, this is extremely wasteful, it's just an example of
* how to use this method. Realistically, most users will never have
* to manually allocate host memory as that is handled by the
* init methods. If memory allocation is successful, allocateHost
 * returns 0*/
    auto err = cuArray->allocateHost();

    /* Check if host memory allocation was successful. */
    if (err == 0) {
        std::cout
                « "Host memory allocated successfully."
                « std::endl;
    } else {
        std::cout
                « "Host memory allocation failed."
                « std::endl;
    }

/* Copy data from device to host. */
    cuArray->toHost();

    delete cuArray;
    return 0;
}
```

argsort()

```
template<typename T >
CuArray< int > * CuArray< T >::argsort (
             int i )
```

Performs an argsort on a specified row of the CuArray. Returns a new CuArray containing sorted indices.

Parameters

i	Column index to argsort.

Returns

Pointer to a new CuArray with sorted indices.

C++ Example

```
#include <cuarray.h>
#include <iostream>

int main() {
    std::cout
            « "Running "
            « __FILE__
            « std::endl;

/* Creates a new float CuArray instance */
    CuArray<float> *cuArray = new CuArray<float>();

/* Initialize the CuArray with 300 rows and 300 columns */
    auto rows = 300;
    auto cols = 300;
    cuArray->init(rows,
                  cols);

/* Fill the CuArray with random values */
    for (int i = 0; i < cuArray->m(); i++) {
        for (int j = 0; j < cuArray->n(); j++) {
            cuArray->host()[i * cuArray->n() + j] =
                    static_cast<float>(rand() / (float) RAND_MAX);
        }
    }

/* Create a new CuArray with indices that sort the 8th row
 * of the original CuArray.*/
    auto cuArrayRowIndex = 7;
    auto sortedIndicesCuArray = cuArray->argsort(cuArrayRowIndex);

/* Create a new CuArray containing sorted data from the 8th row
 * of the original CuArray.*/
    auto sortedCuArray = cuArray->sort(cuArrayRowIndex);

/* Print the sorted CuArray and the corresponding values from the
 * original CuArray using the sortedIndicesCuArray.*/
    for (int j = 0; j < sortedCuArray->n(); j++) {
        auto sortedIndex = sortedIndicesCuArray->get(0,
                                                     j);
        auto sortedValue = sortedCuArray->get(0,
                                              j);
        auto sortedValueFromOriginalCuArray =
                cuArray->get(sortedIndex,
                             cuArrayRowIndex);
        std::cout
                « sortedIndex
                « " "
                « sortedValue
                « " "
                « sortedValueFromOriginalCuArray
                « std::endl;
    }

/* Cleanup time. */
    delete cuArray;
    delete sortedCuArray;
    delete sortedIndicesCuArray;
    return 0;
}
```

Python Example

```
import numpy as np

"""
Always precede CuArray with the data type
Here we are importing the CuArray int and float templates
```

```
"""
from cuarray import FloatCuArray

print("Running", __file__)

"""
Create a new float CuArray instance
"""
float_cuarray = FloatCuArray()

"""
Create a random float32 numpy array with 10 rows
and 10 columns
"""
numpy_array = np.random.rand(10, 10).astype(np.float32)

"""Load the numpy array into the CuArray"""
float_cuarray.fromNumpy2D(numpy_array)

"""
Perform a descending sort on
the 8th row of float_cuarray
"""
sorted_cuarray = float_cuarray.sort(7)

"""
Get the indices that sort the 8th row of float_cuarray
"""
argsort_cuarray = float_cuarray.argsort(7)

"""
Print the sorted 8th row of float_cuarray using
sorted_cuarray and argsort_cuarray indices
"""
for _ in range(10):
    sort_idx = argsort_cuarray[0][_]
    print(
        sorted_cuarray[0][_],
        float_cuarray[7][sort_idx]
    )
```

bytes()

```
template<typename T >
size_t CuArray< T >::bytes ( ) const
```

Returns the total size in bytes of the CuArray data.

Includes both the host and device memory.

Returns

Size in bytes.

C++ Example

```
#include <cuarray.h>
#include <iostream>

int main() {
    std::cout
            « "Running "
            « __FILE__
            « std::endl;
/* Creates a new float CuArray instance */
    CuArray<float> *cuArray = new CuArray<float>();

/*
 * Initializes the CuArray with 10 rows and 5 columns
 * and allocates memory on host.
 */
    cuArray->init(10,
                  5);

/* Get the number of bytes the CuArray data occupies */
```

```
    auto bytes_ = cuArray->bytes();

/* Print the total number of bytes in cuArray. */
    std::cout
            « "Number of bytes: "
            « bytes_
            « std::endl;
/* Output:
 * Number of bytes: 200
 */

    delete cuArray;
    return 0;
}
```

Python Example

deallocateDevice()

```
template<typename T >
CuArrayError CuArray< T >::deallocateDevice ( )
```

Deallocates memory for the device data.

Returns

CuArrayError indicating success or failure of the operation.

C++ Example

deallocateHost()

```
template<typename T >
CuArrayError CuArray< T >::deallocateHost ( )
```

Deallocates memory for the host data.

Returns

CuArrayError indicating success or failure of the operation.

C++ Example

```
#include <cuarray.h>
#include <random>

int main() {

/* Creates a new float CuArray instance */
    CuArray<float> *cuArray = new CuArray<float>();

/* Initialize the CuArray with 300 rows and 300 columns */
    auto rows = 300;
    auto cols = 300;
    cuArray->init(rows,
                  cols);

/* Fill the CuArray with random values */
    for (int i = 0; i < cuArray->m(); i++) {
        for (int j = 0; j < cuArray->n(); j++) {
            cuArray->host()[i * cuArray->n() + j] =
                    static_cast<float>(rand() / (float) RAND_MAX);
```

```
        }
    }
/* Allocate device memory. */
    cuArray->allocateDevice();

/* Copy data from host to device. */
    cuArray->toDevice();

/* Deallocate the host array to reduce memory usage if it's not needed again. */

    cuArray->deallocateHost();

/* Set the number of threads per block to 1024 */
    auto threadsPerBlock = 1024;

/* Set the number of blocks to the ceiling of the number of elements
 * divided by the number of threads per block. */
    auto blocksPerGrid =
            (cuArray->size() + threadsPerBlock - 1) / threadsPerBlock;

/* Launch a CUDA kernel that does something cool and only takes
 * a single float array as an argument
 *«<blocksPerGrid, threadsPerBlock»>kernel(cuArray->device()); */

/* Free device memory. */
    delete cuArray;
    return 0;
}
```

device()

```
template<typename T >
T *& CuArray< T >::device ( )
```

Returns a reference to the device data.

Returns

Reference to the device data.

C++ Example

```
#include <cuarray.h>
#include <iostream>

int main() {
    std::cout
            « "Running "
            « __FILE__
            « std::endl;

/* Creates a new float CuArray instance */
    CuArray<float> *cuArray = new CuArray<float>();

/* Initialize the CuArray with 3 rows and 3 columns */
    cuArray->init(3,
                  3);

/*Set each i, j element equal to i*3 + j */
    for (int i = 0; i < 9; i++) {
        cuArray->host()[i] = i;
    }

/* Allocate device memory. */
    cuArray->allocateDevice();

/* Copy data from host to device. */
    cuArray->toDevice();

/* Set deviceArray equal to cuArray's device data via the
 * device() method, */
    auto deviceArray = cuArray->device();
/* which can be used in CUDA kernels.
 * Eg.) «<1, 1»>kernel(deviceArray)*/


/* delete frees both host and device memory. */
    delete cuArray;
    return 0;
}
```

fromCuArrayDeepCopy()

```
template<typename T >
CuArrayError CuArray< T >::fromCuArrayDeepCopy (
             CuArray< T > * cuArray,
             int start,
             int end,
             int m,
             int n )
```

Performs a deep copy of data from another CuArray within a specified row range. Copies the host data from the given CuArray, including all data within the inclusive range defined by 'start' and 'end'. Memory for the copied data is allocated in this CuArray's host memory.

Parameters

cuArray	Pointer to the source CuArray.
start	Index of the first row to copy.
end	Index of the last row to copy.
m	Number of rows in this CuArray.
n	Number of columns in this CuArray.

Returns

CuArrayError indicating the operation's success or failure.

C++ Example

```
#include <cuarray.h>
#include <iostream>

int main() {
    std::cout
            « "Running "
            « __FILE__
            « std::endl;

/* Create a new float CuArray instance */
    auto cuArray = new CuArray<float>;

/* Initialize the CuArray with 3 rows and 3 columns */
    cuArray->init(3,
                  3);

/*Set each i, j element equal to i*3 + j */
    for (int i = 0; i < 9; i++) {
        cuArray->host()[i] = i;
    }

/*
 * Create a float 'CuArray' that
 * will be a deep copy of the last two cuArray rows
 */
    auto cuArray2x3Copy = new CuArray<float>;
    cuArray2x3Copy->init(2,
                         3);

/* First row to copy from cuArray into cuArray2x3Copy */
    int startRowIndex = 1;

/* Last row to copy from cuArray into cuArray2x3Copy */
    int endRowIndex = 2;

    cuArray2x3Copy->fromCuArrayDeepCopy(
            cuArray, /*Source for copying data into cuArray2x3Copy. This method is
            * significantly safer than its shallow copy equivalent. However, it is also
            * slower, which can impact performance if it's called a lot.*/
            startRowIndex, /* First row to copy from cuArray into cuArray2x3Copy */
```

```
            endRowIndex, /* Last row to copy from cuArray into cuArray2x3Copy */
            cuArray2x3Copy->m(), /* Number of rows in cuArray2x3Copy */
            cuArray2x3Copy->n() /* Number of columns in cuArray2x3Copy */
    );

/* Print each element in cuArray2x3Copy */
    for (int i = 0; i < cuArray2x3Copy->m(); i++) {
        for (int j = 0; j < cuArray2x3Copy->n(); j++) {
            std::cout << cuArray2x3Copy->get(i,
                                             j) << " ";
        }
        std::cout << std::endl;
    }
/* Output:
 * 3 4 5
 * 6 7 8
 */

/* Both cuArray and cuArray2x3Copy own their data.*/
    std::cout
            << cuArray->owner() << " "
            << cuArray2x3Copy->owner()
            << std::endl;
/* Output:
 * 1 1
 */

    delete cuArray2x3Copy;
    delete cuArray;
    return 0;
}
```

Python Example

```
"""
Always precede CuArray with the data type
Here we are importing float templates.
"""
from cuarray import FloatCuArray

import numpy as np

print("Running", __file__)

"""Create two new float CuArray instances"""
float_cuarray1 = FloatCuArray()
float_cuarray2 = FloatCuArray()

"""Initialize float_cuarray1 with 10 rows and 10 columns"""
float_cuarray1.init(10, 10)

"""Fill float_cuarray1 with random values"""
for i in range(float_cuarray1.m()):
    for j in range(float_cuarray1.n()):
        val = np.random.random()
        float_cuarray1[i][j] = val

"""Copy the data from float_cuarray1 into float_cuarray2"""
float_cuarray2.fromCuArray(float_cuarray1, 0, 9, 10, 10)

"""
Print both CuArrays. Also this performs a deep copy for
memory safety.
"""
for i in range(float_cuarray1.m()):
    for j in range(float_cuarray1.n()):
        print(float_cuarray1[i][j], float_cuarray2[i][j])
```

fromCuArrayShallowCopy()

```
template<typename T >
CuArrayError CuArray< T >::fromCuArrayShallowCopy (
             CuArray< T > * cuArray,
             int start,
             int end,
             int m,
             int n )
```

Performs a shallow copy of data from another CuArray within a specified row range. Copies the host data from the given CuArray, within the inclusive range specified by 'start' and 'end'. This CuArray does not own the copied data, and deallocation is handled by the source CuArray.

Parameters

cuArray	Pointer to the source CuArray.
start	Index of the first row to copy.
end	Index of the last row to copy.
m	Number of rows in this CuArray.
n	Number of columns in this CuArray.

Returns

CuArrayError indicating the operation's success or failure.

C++ Example

```
#include <cuarray.h>
#include <iostream>

int main() {
    std::cout
            « "Running "
            « __FILE__
            « std::endl;

/* Create a new float CuArray instance */
    auto cuArray = new CuArray<float>;

/* Initialize the CuArray with 3 rows and 3 columns */
    cuArray->init(3,
                  3);

/*Set each i, j element equal to i*3 + j */
    for (int i = 0; i < 9; i++) {
        cuArray->host()[i] = i;
    }

/*
 * Create a float 'CuArray' that
 * will be a shallow copy of the last two cuArray rows
 */
    auto cuArray2x3Copy = new CuArray<float>;
    cuArray2x3Copy->init(2,
                         3);

/* First row to copy from cuArray into cuArray2x3Copy */
    int startRowIndex = 1;

/* Last row to copy from cuArray into cuArray2x3Copy */
    int endRowIndex = 2;

    cuArray2x3Copy->fromCuArrayShallowCopy(
            cuArray, /* Source for copying data into cuArray2x3Copy.
            * Both cuArray and cuArray2x3Copy will point to the same
            * data, which helps with
            * performance at the expense of being extremely dangerous. As an
            * attempt to make this method somewhat safe, there is an "owner"
            * attribute that is set to 1 if the CuArray owns the data and 0
            * otherwise. Logic is implemented in the destructor to check for ownership
            * and only delete data if the CuArray owns the data. As of now, this method has
            * passed all real life stress tests, and CUDA-MEMCHECK doesn't hate it,
            * but it still shouldn't be used in the vast majority of cases.
            * The legitimate reason this should ever be called is when you have to
            * pass the CuArray data as a double pointer to a function that
            * cannot itself take a CuArray object. Eg.) A CUDA kernel.*/
            startRowIndex, /* First row to copy from cuArray into cuArray2x3Copy */
            endRowIndex, /* Last row to copy from cuArray into cuArray2x3Copy */
            cuArray2x3Copy->m(), /* Number of rows in cuArray2x3Copy */
            cuArray2x3Copy->n() /* Number of columns in cuArray2x3Copy */
    );

/* Print each element in cuArray2x3Copy */
```

```
    for (int i = 0; i < cuArray2x3Copy->m(); i++) {
        for (int j = 0; j < cuArray2x3Copy->n(); j++) {
            std::cout << cuArray2x3Copy->get(i,
                                             j) << " ";
        }
        std::cout << std::endl;
    }
/* Output:
 * 3 4 5
 * 6 7 8
 */
    delete cuArray2x3Copy;
    delete cuArray;
    return 0;
}
```

fromNumpy() [1/2]

```
template<typename T >
CuArrayError CuArray< T >::fromNumpy (
            T * NUMPY_ARRAY,
            int NUMPY_ARRAY_DIM1 )
```

Copies data from a NumPy array to the CuArray.

Parameters

NUMPY_ARRAY	Pointer to input NumPy array.
NUMPY_ARRAY_DIM1	Dimension 1 of the NumPy array.

Returns

CuArrayError indicating success or failure of the operation.

C++ Example

```
#include <cuarray.h>
#include <iostream>
#include <random>

int main() {
    std::cout
            << "Running "
            << __FILE__
            << std::endl;

/* Creates a new float CuArray instance */
    CuArray<float> *cuArray = new CuArray<float>();

/* Create a float vector with 10 elements.*/
    auto *NUMPY_ARRAY = new float[10];
    int rows = 10;

/* Fill the NUMPY_ARRAY with random values */
    for (int i = 0; i < rows; i++) {
        NUMPY_ARRAY[i] =
                (float) rand() / (float) RAND_MAX;
    }

/* Copy the NUMPY_ARRAY data into the CuArray. The
 * CuArray has the same dimensions as the NUMPY_ARRAY. */
    cuArray->fromNumpy(
            NUMPY_ARRAY,
            rows
    );

/* Print the CuArray. */
    for (int i = 0; i < rows; i++) {
        std::cout
                << cuArray->host()[i]
```

```
                « " ";
    }
    std::cout
            « std::endl;


/* Free the NUMPY_ARRAY and CuArray. */
    delete cuArray;
    delete[] NUMPY_ARRAY;
    return 0;
}
```

Python Example

```
import numpy as np

"""
Always precede CuArray with the data type
Here we are importing the CuArray float template
"""
from cuarray import FloatCuArray

print("Running", __file__)

"""Create a new float CuArray instance"""
float_cuarray = FloatCuArray()

"""
Create a random float32, 1-dimension numpy array,
with 10 elements
"""

np_array = np.random.rand(10).astype(np.float32)

"""Copy the numpy array to the CuArray instance"""
float_cuarray.fromNumpy1D(np_array)

"""Print the CuArray and numpy array to compare."""
for _ in range(10):
    print(float_cuarray[0][_], np_array[_])
```

fromNumpy() [2/2]

```
template<typename T >
CuArrayError CuArray< T >::fromNumpy (
              T * NUMPY_ARRAY,
              int NUMPY_ARRAY_DIM1,
              int NUMPY_ARRAY_DIM2 )
```

Copies data from a NumPy array to the CuArray.

Parameters

NUMPY_ARRAY	Pointer to the input NumPy array.
NUMPY_ARRAY_DIM1	Dimension 1 of the NumPy array.
NUMPY_ARRAY_DIM2	Dimension 2 of the NumPy array.

Returns

CuArrayError indicating success or failure of the operation.

C++ Example

```
#include <cuarray.h>
#include <iostream>

int main() {
```

```
    std::cout
            « "Running "
            « __FILE__
            « std::endl;

/* Creates a new float CuArray instance */
    CuArray<float> *cuArray = new CuArray<float>();

/* Create a linear float array that has 10 rows and 10 columns.*/
    auto *NUMPY_ARRAY = new float[100];
    int rows = 10;
    int cols = 10;

/* Fill the NUMPY_ARRAY with random values */
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            NUMPY_ARRAY[i * cols + j] =
                    (float) rand() / (float) RAND_MAX;
        }
    }

/* Copy the NUMPY_ARRAY data into the CuArray. The
 * CuArray has the same dimensions as the NUMPY_ARRAY. */
    cuArray->fromNumpy(
            NUMPY_ARRAY,
            rows,
            cols
    );

/* Print the CuArray. */
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            std::cout
                    « cuArray->host()[i * cols + j]
                    « " ";
        }
        std::cout
                « std::endl;
    }
    std::cout
            « std::endl;


/* Free the NUMPY_ARRAY and CuArray. */
    delete cuArray;
    delete[] NUMPY_ARRAY;
    return 0;
}
```

Python Example

```
import numpy as np
"""
Always precede CuArray with the data type
Here we are importing the CuArray float template
"""
from cuarray import FloatCuArray

print("Running", __file__)

"""Create a new float CuArray instance"""
float_cuarray = FloatCuArray()

"""
Create a random float32, 2-dimension numpy array
with 10 rows and 10 columns.
"""
np_array = np.random.random((10, 10)).astype(np.float32)

"""Copy the numpy array to the CuArray instance"""
float_cuarray.fromNumpy2D(np_array)

"""Print the CuArray and numpy array to compare."""
for i in range(10):
    for j in range(10):
        print(float_cuarray[i][j], np_array[i][j])
```

get()

```
template<typename T >
T CuArray< T >::get (
```

```
            int i,
            int j ) const
```

Returns the value at a specified position in the CuArray.

Parameters

i	Row index.
j	Column index.

Returns

Value at the specified position.

C++ Example

```
#include "cuarray.h"
#include <iostream>
#include <random>

int main() {
    std::cout
            « "Running "
            « __FILE__
            « std::endl;

/* Creates a new float CuArray instance that will have 10 rows
 * and 10 columns*/
    CuArray<float> *cuArray = new CuArray<float>();
    int m = 10; /* Number of rows */
    int n = 10; /* Number of columns */
    cuArray->init(m,
                  n);

/* Fill the CuArray with random values */
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            cuArray->set((float) rand() / (float) RAND_MAX,
                         i,
                         j);
        }
    }

/* As it's name implies, get(i, j) returns the value at the
 * specified position (i, j) in the CuArray. */

/* Use the get method to print the value at each position in the CuArray. */
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            std::cout
                    « cuArray->get(i,
                                   j)
                    « " ";
        }
        std::cout
                « std::endl;
    }

/* Free the CuArray. */
    delete cuArray;
    return 0;
}
```

Python Example

```
import numpy as np

"""
Always precede CuArray with the data type
Here we are importing the CuArray float template
"""
from cuarray import FloatCuArray

print("Running", __file__)

"""
```

```
Create a new float CuArray instance with
10 rows and 10 columns
"""
float_cuarray = FloatCuArray()
float_cuarray.init(10, 10)

"""Fill the array with random values"""

for i in range(10):
    for j in range(10):
        val = np.random.random()
        float_cuarray.set(val, i, j)

"""Print the array"""
print(float_cuarray)
```

host()

```
template<typename T >
T *& CuArray< T >::host ( )
```

Returns a reference to the host data.

Returns

Reference to the host data.

C++ Example

```
#include <cuarray.h>
#include <iostream>

int main() {
    std::cout
            « "Running "
            « __FILE__
            « std::endl;
/* Creates a new float CuArray instance */
    CuArray<float> *cuArray = new CuArray<float>();

/* Initialize the CuArray with 3 rows and 3 columns */
    cuArray->init(3,
                  3);

/*Set each i, j element equal to i*3 + j */
    for (int i = 0; i < 9; i++) {
        cuArray->host()[i] = i;
    }

/* Print each element in cuArray's host memory.
 * The host data is linear and stored in row major order. To
 * access element i,j you would use the linear index
 * i*n+j, where n is the number of columns.*/
    for (int i = 0; i < cuArray->m(); i++) {
        for (int j = 0; j < cuArray->n(); j++) {
            std::cout « cuArray->host()[i * cuArray->n() + j] « " ";
        }
        std::cout « std::endl;
    }
/* Output:
 * 0 1 2
 * 3 4 5
 * 6 7 8
 */

    delete cuArray;
    return 0;
}
```

init() [1/2]

```
template<typename T >
CuArrayError CuArray< T >::init (
             int m,
             int n )
```

Initializes CuArray with specified dimensions and allocates memory on host and device.

Parameters

m	Number of rows.
n	Number of columns.

Returns

CuArrayError indicating operation success or failure.

C++ Example

```
#include <iostream>
#include "cuarray.h"

int main() {
    std::cout
            « "Running "
            « __FILE__
            « std::endl;

/* Creates a new float CuArray instance */
    auto *cuArray = new CuArray<float>();

/*
 * Initializes the CuArray with 10 rows and 5 columns
 * and allocates memory on host.
 */
    cuArray->init(10,
                  5);

    /* Print the cuArray */
    for (int i = 0; i < cuArray->m(); i++) {
        for (int j = 0; j < cuArray->n(); j++) {
            std::cout « cuArray->get(i,
                                     j) « " ";
        }
        std::cout « std::endl;
    }

/* Free the memory allocated on host and device */
    delete cuArray;

    return 0;
}
```

Python Example

```
"""
Always precede CuArray with the data type
Here we are importing float templates.
"""
from cuarray import FloatCuArray

print("Running", __file__)

"""Create a new float CuArray instance"""
float_cuarray = FloatCuArray()

"""Initialize the float CuArray with 10 rows and 10 columns"""
float_cuarray.init(10, 10)

"""
Print the CuArray,
which has a __repr__ method implemented in the SWIG interface
"""
print(float_cuarray)
```

init() [2/2]

```
template<typename T >
CuArrayError CuArray< T >::init (
             T * host,
```

```
        int m,
        int n )
```

Initializes CuArray with specified host data and dimensions, performing a shallow copy. Allocates memory on both the host and the device. The data is shallow copied, so the ownership remains unchanged.

Parameters

host	Pointer to input host data.
m	Number of rows.
n	Number of columns.

Returns

CuArrayError indicating operation success or failure.

C++ Example

```
#include "cuarray.h"
#include <iostream>

int main() {
    std::cout
            « "Running "
            « __FILE__
            « std::endl;

/* Creates a new float CuArray instance */
    CuArray<float> *cuArray = new CuArray<float>();

/*
 * Initializes the CuArray with 10 rows and 5 columns
 * and allocates memory on host.
 */
    cuArray->init(10,
                  5);

/* Create a 50-element float vector and fill it with random values */
    auto a = new float[50];
    for (int i = 0; i < 50; i++) {
        a[i] = static_cast<float>(rand() / (float) RAND_MAX);
    }

/* Initialize the CuArray with data from "a", preserving
 * overall size while setting new dimensions
 * (similar to NumPy's reshape method). */
    cuArray->init(a,
                  10,
                  5);

/* Print each element in cuArray's host memory.
 * The host data is linear and stored in row major order. To
 * access element i,j you would use the linear index
 * i*n+j, where n is the number of columns.*/
    for (int i = 0; i < cuArray->m(); i++) {
        for (int j = 0; j < cuArray->n(); j++) {
            std::cout « cuArray->get(i,
                                     j) « " ";
            std::cout « a[i * cuArray->n() + j] « std::endl;
        }
        std::cout « std::endl;
    }

/* Delete "a" and cuArray */
    delete[] a;
    delete cuArray;
    return 0;
}
```

load()

```
template<typename T >
CuArrayError CuArray< T >::load (
             const std::string & fname )
```

Loads CuArray data from a specified file.

Parameters

fname	File name to load from.

Returns

CuArrayError indicating success or failure of the operation.

C++ Example

```cpp
#include "cuarray.h"
#include <iostream>
#include <random>

int main() {
    std::cout
            « "Running "
            « __FILE__
            « std::endl;

/* Create a new double CuArray instance. We're using a double vs. float
 * here because the numpy array is a float64 array. If you tried
 * to load this file into a CuArray<float> it would cause a
 * segmentation fault.*/
    CuArray<double> *cuArray = new CuArray<double>();

/*
 * Load a serialized numpy array with 2000 elements from the C++ test data directory.
 * NETSCI_ROOT_DIR, used here, is defined in CMakeLists. Ignore warnings in IDEs
 * about it being undefined; it's a known issue and does not affect functionality.
 */
    auto npyFname = NETSCI_ROOT_DIR
            "/tests/netcalc/cpp/data/2X_1D_1000_4.npy";
    cuArray->load(npyFname);

/* Print the CuArray. */
    for (int i = 0; i < cuArray->m(); i++) {
        for (int j = 0; j < cuArray->n(); j++) {
            std::cout
                    « cuArray->get(i,
                                   j)
                    « std::endl;
        }
    }

/* Free the CuArray. */
    delete cuArray;
    return 0;
}
```

Python Example

```python
import numpy as np

"""
Always precede CuArray with the data type
Here we are importing the CuArray float template
"""
from cuarray import FloatCuArray

print("Running", __file__)


"""
Create a new float CuArray instance with
10 rows and 10 columns
"""
float_cuarray = FloatCuArray()

"""
Create a random float32 numpy array with 10 rows
and 10 columns
"""
numpy_array = np.random.rand(10, 10).astype(np.float32)

"""Save the numpy array to a .npy file"""
np.save("tmp.npy", numpy_array)

"""
```

```
Load the .npy file into the float CuArray instance
"""
float_cuarray.load("tmp.npy")

"""Print the CuArray and the numpy array"""
for i in range(10):
    for j in range(10):
        print(float_cuarray[i][j], numpy_array[i, j])
```

m()

```
template<typename T >
int CuArray< T >::m ( ) const
```

Returns the number of rows in the CuArray.

Returns

Number of rows.

C++ Example

```
#include <cuarray.h>
#include <iostream>

int main() {
    std::cout
            « "Running "
            « __FILE__
            « std::endl;
/* Creates a new float CuArray instance */
    CuArray<float> *cuArray = new CuArray<float>();

/*
 * Initializes the CuArray with 10 rows and 5 rows
 * and allocates memory on host.
 */
    cuArray->init(10,
                  5);

/* Get the number of rows in the CuArray */
    int m = cuArray->m();

/* Print the number of rows */
    std::cout
            « "Number of rows: "
            « m
            « std::endl;
/* Output:
 * Number of rows: 10
 */

    delete cuArray;
    return 0;
}
```

Python Example

```
import numpy as np
"""
Always precede CuArray with the data type
Here we are importing float template.
"""
from cuarray import FloatCuArray

print("Running", __file__)


"""Create a new float CuArray instance"""
float_cuarray = FloatCuArray()

"""Initialize the float CuArray with 10 rows and 2 columns"""
float_cuarray.init(10, 2)

"""Print the number of rows in the CuArray"""
print(float_cuarray.m())
```

n()

```
template<typename T >
int CuArray< T >::n ( ) const
```

Returns the number of columns in the CuArray.

Returns

Number of columns.

C++ Example

```
#include <cuarray.h>
#include <iostream>

int main() {
    std::cout
            « "Running "
            « __FILE__
            « std::endl;
/* Creates a new float CuArray instance */
    CuArray<float> *cuArray = new CuArray<float>();

/*
 * Initializes the CuArray with 10 rows and 5 columns
 * and allocates memory on host.
 */
    cuArray->init(10,
                  5);

/* Get the number of columns in the CuArray */
    int n = cuArray->n();

/* Print the number of columns */
    std::cout
            « "Number of columns: "
            « n
            « std::endl;
/* Output:
 * Number of columns: 5
 */

    delete cuArray;
    return 0;
}
```

Python Example

```
"""
Always precede CuArray with the data type
Here we are importing the float template.
"""
from cuarray import FloatCuArray

print("Running", __file__)

"""Create a new float CuArray instance"""
float_cuarray = FloatCuArray()

"""Initialize the float CuArray with 10 rows and 2 columns"""
float_cuarray.init(10, 2)

"""Print the number of columns in the CuArray"""
print(float_cuarray.n())
```

operator[]()

```
template<typename T >
T & CuArray< T >::operator[] (
            int i ) const
```

Returns a reference to the element at a specified index in the CuArray.

Parameters

i	Index of the element.

Returns

Reference to the element at the specified index.

C++ Example

```cpp
#include <cuarray.h>
#include <iostream>

int main() {
    std::cout
            « "Running "
            « __FILE__
            « std::endl;

/* Create a new float CuArray instance */
    auto cuArray = new CuArray<float>;

/* Initialize the CuArray with 3 rows and 3 columns */
    cuArray->init(3,
                  3);

/*Set each i, j element equal to i*3 + j */
    for (int i = 0; i < 9; i++) {
        cuArray->host()[i] = i;
    }

/* Calculate the linear index that
 * retrieves the 3rd element in the 2nd row of the CuArray. */
    int i = 1;
    int j = 2;
    int linearIndex = i * cuArray->n() + j;
    auto ijLinearVal = (*(cuArray))[linearIndex];
    auto ijVal = cuArray->get(i,
                              j);

/* Print the values at the linear index and the (i, j) index. */
    std::cout
            « ijLinearVal
            « " "
            « ijVal
            « std::endl;

/*Deallocate memory*/
    delete cuArray;
    return 0;
}
```

Python Example

```python
import numpy as np

"""
Always precede CuArray with the data type
Here we are importing the CuArray float template
"""
from cuarray import FloatCuArray

print("Running", __file__)

"""
Create a new float CuArray instance
with 10 rows and 10 columns.
"""
float_cuarray = FloatCuArray()
float_cuarray.init(10, 10)

"""Fill it with random values"""
for i in range(10):
    for j in range(10):
        val = np.random.rand()
        float_cuarray.set(val, i, j)

"""Print the 8th row"""
print(float_cuarray[7])

"""Print the 5th element of the 8th row"""
print(float_cuarray[7][4])
```

owner()

```
template<typename T >
int CuArray< T >::owner ( ) const
```

Returns the owner status of the CuArray. Indicates whether the CuArray is responsible for memory deallocation.

Returns

Owner status of the CuArray.

C++ Example

```
#include <cuarray.h>
#include <iostream>

int main() {
    std::cout
            « "Running "
            « __FILE__
            « std::endl;

/* Create a new float CuArray instance */
    auto cuArray = new CuArray<float>;

/* Initialize the CuArray with 3 rows and 3 columns */
    cuArray->init(3,
                  3);

/*Set each i, j element equal to i*3 + j */
    for (int i = 0; i < 9; i++) {
        cuArray->host()[i] = i;
    }

/*
 * Create a float 'CuArray' that
 * will be a shallow copy of the last two cuArray rows
 */
    auto cuArray2x3Copy = new CuArray<float>;
    cuArray2x3Copy->init(2,
                         3);

/* First row to copy from cuArray into cuArray2x3Copy */
    int startRowIndex = 1;

/* Last row to copy from cuArray into cuArray2x3Copy */
    int endRowIndex = 2;

    cuArray2x3Copy->fromCuArrayShallowCopy(
            cuArray, /* Source for copying data into cuArray2x3Copy. See
                  * CuArray::fromCuArrayShallowCopy for more info. */
            startRowIndex, /* First row to copy from cuArray into cuArray2x3Copy */
            endRowIndex, /* Last row to copy from cuArray into cuArray2x3Copy */
            cuArray2x3Copy->m(), /* Number of rows in cuArray2x3Copy */
            cuArray2x3Copy->n() /* Number of columns in cuArray2x3Copy */
    );

/* Now make another CuArray that is a deep copy of cuArray2x3Copy */
    auto cuArray2x3DeepCopy = new CuArray<float>;
    cuArray2x3DeepCopy->init(2,
                             3);
    cuArray2x3DeepCopy->fromCuArrayDeepCopy(
            cuArray, /* Source for copying data into cuArray2x3DeepCopy. See
                  * CuArray::fromCuArrayDeepCopy for more info. */
            startRowIndex, /* First row to copy from cuArray into cuArray2x3DeepCopy */
            endRowIndex, /* Last row to copy from cuArray into cuArray2x3DeepCopy */
            cuArray2x3DeepCopy->m(), /* Number of rows in cuArray2x3DeepCopy */
            cuArray2x3DeepCopy->n() /* Number of columns in cuArray2x3DeepCopy */
    );

/* Check if cuArray2x3Copy owns the host data. */
    auto cuArray2x3CopyOwnsHostData = cuArray2x3Copy->owner();

/* Check if cuArray2x3DeepCopy owns the host data.
 * Sorry for the verbosity :), I'm sure this is painful for
 * Python devs to read (though Java devs are probably loving it).*/
    auto cuArray2x3DeepCopyOwnsHostData = cuArray2x3DeepCopy->owner();

/* Print data in both arrays. */
```

```
    for (int i = 0; i < cuArray2x3Copy->m(); i++) {
        for (int j = 0; j < cuArray2x3Copy->n(); j++) {
            std::cout
                    « cuArray2x3Copy->get(i,
                                          j)
                    « " "
                    « cuArray2x3DeepCopy->get(i,
                                              j)
                    « std::endl;
        }
    }

/* Print ownership info. */
    std::cout
            « "cuArray2x3Copy owns host data: "
            « cuArray2x3CopyOwnsHostData
            « " cuArray2x3DeepCopy owns host data: "
            « cuArray2x3DeepCopyOwnsHostData
            « std::endl;

    delete cuArray2x3Copy;
    delete cuArray2x3DeepCopy;
    delete cuArray;
    return 0;
}
```

save()

```
template<typename T >
void CuArray< T >::save (
              const std::string & fname )
```

Saves CuArray data to a specified file.

Parameters

fname	File name to save to.

C++ Example

```
#include "cuarray.h"
#include <iostream>

#define NETSCI_ROOT_DIR "."

int main() {
    std::cout
            « "Running "
            « __FILE__
            « std::endl;

/* Create a new double CuArray instance that will have 10 rows and 10
* columns*/
    CuArray<float> *cuArray = new CuArray<float>();
    cuArray->init(10,
                  10
    );

/* Fill the CuArray with random values. */
    for (int i = 0; i < cuArray->m(); i++) {
        for (int j = 0; j < cuArray->n(); j++) {
            float val = static_cast <float> (rand()) /
                        static_cast <float> (RAND_MAX);
            cuArray->set(val,
                         i,
                         j);
        }
    }

/* Save the CuArray to a .npy file. */
    auto npyFname = NETSCI_ROOT_DIR "/tmp.npy";
    cuArray->save(npyFname);

/* Create a new CuArray instance from the .npy file. */
    auto cuArrayFromNpy = new CuArray<float>();
```

```
    cuArrayFromNpy->load(npyFname);

/*Print (i, j) elements of the CuArray's next to each other.
 * and check for equality*/
    for (int i = 0; i < cuArray->m(); i++) {
        for (int j = 0; j < cuArray->n(); j++) {
            auto val1 = cuArray->get(i,
                                     j);
            auto val2 = cuArrayFromNpy->get(i,
                                            j);
            bool equal = val1 == val2;
            std::cout
                    << val1
                    << " "
                    << val2
                    << " "
                    << equal
                    << std::endl;
            if (!equal) {
                std::cout
                        << "Values at ("
                        << i
                        << ", "
                        << j
                        << ") are not equal."
                        << std::endl;
                return 1;
            }


        }
    }
    delete cuArray;
    return 0;
}
```

Python Example

```
import numpy as np

"""
Always precede CuArray with the data type
Here we are importing the CuArray float template
"""
from cuarray import FloatCuArray

print("Running", __file__)

"""
Create a new float CuArray instance with
10 rows and 10 columns
"""
float_cuarray = FloatCuArray()

"""
Create a random float32 numpy array with 10 rows
and 10 columns
"""
numpy_array = np.random.rand(10, 10).astype(np.float32)

"""Save the numpy array to a .npy file"""
np.save("tmp.npy", numpy_array)

"""
Load the .npy file into the float CuArray instance
"""
float_cuarray.load("tmp.npy")

"""Print the CuArray and the numpy array"""
for i in range(10):
    for j in range(10):
        print(float_cuarray[i][j], numpy_array[i, j])
```

set()

```
template<typename T >
CuArrayError CuArray< T >::set (
            T value,
            int i,
            int j )
```

Sets a value at a specified position in the CuArray.

Parameters

value	Value to set.
i	Row index.
j	Column index.

Returns

CuArrayError indicating success or failure of the operation.

C++ Example

```cpp
#include "cuarray.h"
#include <iostream>

int main() {
    std::cout
            « "Running "
            « __FILE__
            « std::endl;
/* Creates a new float CuArray instance that will have 10 rows
 * and 10 columns*/
    CuArray<float> *cuArray = new CuArray<float>();
    int m = 10; /* Number of rows */
    int n = 10; /* Number of columns */
    cuArray->init(m,
                  n);

/* As it's name implies, set(value, i, j) sets the value at the
 * specified position (i, j) in the CuArray. */

/* Use the set method to set the value at each position in the CuArray
 * to a random number.*/
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            cuArray->set((float) rand() / (float) RAND_MAX,
                         i,
                         j);
        }
    }

/* Print the CuArray. */
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            std::cout
                    « cuArray->get(i,
                                   j)
                    « " ";
        }
        std::cout
                « std::endl;
    }

/* Free the CuArray. */
    delete cuArray;
    return 0;
}
```

Python Example

```python
import numpy as np

"""
Always precede CuArray with the data type
Here we are importing the CuArray float template
"""
from cuarray import FloatCuArray

print("Running", __file__)


"""
Create a new float CuArray instance with
10 rows and 10 columns
"""
float_cuarray = FloatCuArray()
```

```
float_cuarray.init(10, 10)

"""Fill the array with random values"""

for i in range(10):
    for j in range(10):
        val = np.random.random()
        float_cuarray.set(val, i, j)

"""Print the array using the get method"""
for i in range(10):
    for j in range(10):
        val = float_cuarray.get(i, j)
        print('{0:.{1}f}'.format(val, 5), end=" ")
    print()
```

size()

```
template<typename T >
int CuArray< T >::size ( ) const
```

Returns the total number of elements in the CuArray.

Returns

Total number of elements (rows multiplied by columns).

C++ Example

```
#include <cuarray.h>
#include <iostream>

int main() {
    std::cout
            « "Running "
            « __FILE__
            « std::endl;
/* Creates a new float CuArray instance */
    CuArray<float> *cuArray = new CuArray<float>();

/*
 * Initializes the CuArray with 10 rows and 5 columns
 * and allocates memory on host.
 */
    cuArray->init(10,
                  5);

/* Get the total number of values in the CuArray */
    int size = cuArray->size();

/* Print the total number of values in cuArray. */
    std::cout
            « "Number of values: "
            « size
            « std::endl;
/* Output:
 * Number of values: 50
 */

    delete cuArray;
    return 0;
}
```

Python Example

```
"""
Always precede CuArray with the data type
Here we are importing the float template.
"""
from cuarray import FloatCuArray

print("Running", __file__)

"""Create a new float CuArray instance"""
float_cuarray = FloatCuArray()

"""Initialize the float CuArray with 10 rows and 2 columns"""
float_cuarray.init(10, 2)

"""Print the total number of values in the CuArray"""
print(float_cuarray.size())
```

sort()

```
template<typename T >
CuArray< T > * CuArray< T >::sort (
            int i )
```

Sorts CuArray based on the values in a specified row.

Parameters

i	Index of the row to sort by.

Returns

Pointer to a new CuArray with sorted data.

C++ Example

```
#include <cuarray.h>
#include <random>
#include <iostream>

int main() {
    std::cout
            « "Running "
            « __FILE__
            « std::endl;
/* Creates a new float CuArray instance */
    CuArray<float> *cuArray = new CuArray<float>();

/* Initialize the CuArray with 300 rows and 300 columns */
    auto rows = 300;
    auto cols = 300;
    cuArray->init(rows,
                  cols);

/* Fill the CuArray with random values */
    for (int i = 0; i < cuArray->m(); i++) {
        for (int j = 0; j < cuArray->n(); j++) {
            cuArray->host()[i * cuArray->n() + j] =
                    static_cast<float>(rand() / (float) RAND_MAX);
        }
    }

/* Create a new CuArray that contains the sorted data from the
 * 8th row of the original CuArray. */
    auto sortedCuArray = cuArray->sort(7);

/* Print the sorted CuArray. */
    for (int j = 0; j < sortedCuArray->n(); j++) {
        std::cout
                « sortedCuArray->get(0,
                                     j)
                « std::endl;
    }

/* Cleanup time. */
    delete cuArray;
    delete sortedCuArray;
    return 0;
}
```

Python Example

```
import numpy as np

"""
Always precede CuArray with the data type
Here we are importing the CuArray float template
"""
from cuarray import FloatCuArray

print("Running", __file__)

"""
```

```
Create a new float CuArray instance
"""
float_cuarray = FloatCuArray()

"""
Create a random float32 numpy array with 10 rows
and 10 columns
"""
numpy_array = np.random.rand(10, 10).astype(np.float32)

"""Load the numpy array into the CuArray"""
float_cuarray.fromNumpy2D(numpy_array)

"""
Perform an out of place descending sort on the
8th column of float_cuarray
"""
sorted_cuarray = float_cuarray.sort(7)

"""
Print the 8th row of the original
CuArray and sorted_cuarray
"""
print(sorted_cuarray)
print(float_cuarray[7])
```

toDevice()

```
template<typename T >
CuArrayError CuArray< T >::toDevice ( )
```

Copies data from the host to the device.

Returns

CuArrayError indicating success or failure of the operation.

C++ Example

```
#include <cuarray.h>
#include <random>
#include <iostream>

int main() {
    std::cout
            « "Running "
            « __FILE__
            « std::endl;
/* Creates a new float CuArray instance */
    CuArray<float> *cuArray = new CuArray<float>();

/* Initialize the CuArray with 300 rows and 300 columns */
    auto rows = 300;
    auto cols = 300;
    cuArray->init(rows,
                  cols);

/* Fill the CuArray with random values */
    for (int i = 0; i < cuArray->m(); i++) {
        for (int j = 0; j < cuArray->n(); j++) {
            cuArray->host()[i * cuArray->n() + j] =
                    static_cast<float>(rand() / (float) RAND_MAX);
        }
    }
/* Allocate device memory. */
    cuArray->allocateDevice();

/* Copy data from host to device. */
    cuArray->toDevice();

/* Frees host and device memory. */
    delete cuArray;
    return 0;
}
```

toHost()

```
template<typename T >
CuArrayError CuArray< T >::toHost ( )
```

Copies data from the device to the host.

Returns

CuArrayError indicating success or failure of the operation.

C++ Example

```
#include <cuarray.h>
#include <iostream>

int main() {
    std::cout
            « "Running "
            « __FILE__
            « std::endl;

/* Creates a new float CuArray instance */
    CuArray<float> *cuArray = new CuArray<float>();

/* Initialize the CuArray with 300 rows and 300 columns */
    auto rows = 300;
    auto cols = 300;
    cuArray->init(rows,
                  cols);

/* Fill the CuArray with random values */
    for (int i = 0; i < cuArray->m(); i++) {
        for (int j = 0; j < cuArray->n(); j++) {
            cuArray->host()[i * cuArray->n() + j] =
                    static_cast<float>(rand() / (float) RAND_MAX);
        }
    }
/* Allocate device memory. */
    cuArray->allocateDevice();

/* Copy data from host to device. */
    cuArray->toDevice();

/* Set the number of threads per block to 1024 */
    auto threadsPerBlock = 1024;

/* Set the number of blocks to the ceiling of the number of elements
 * divided by the number of threads per block. */
    auto blocksPerGrid =
            (cuArray->size() + threadsPerBlock - 1) / threadsPerBlock;

/* Launch a CUDA kernel that does something cool and only takes
 * a single float array as an argument
 *«<blocksPerGrid, threadsPerBlock»>kernel(cuArray->device()); */

/* Copy data from device to host. */
    cuArray->toHost();

/* Frees host and device memory. */
    delete cuArray;
    return 0;
}
```

toNumpy() [1/2]

```
template<typename T >
void CuArray< T >::toNumpy (
             T ** NUMPY_ARRAY,
             int ** NUMPY_ARRAY_DIM1 )
```

Copies data from the CuArray to a NumPy array.

Parameters

NUMPY_ARRAY	Pointer to output NumPy array.
NUMPY_ARRAY_DIM1	Dimension 1 of the NumPy array.

C++ Example

```
#include "cuarray.h"
#include <iostream>
#include <random>

int main() {
    std::cout
            « "Running "
            « __FILE__
            « std::endl;
/* Creates a new float CuArray instance 1 row and 10 columns*/
    CuArray<float> *cuArray = new CuArray<float>();
    int m = 1; /* Number of rows */
    int n = 10; /* Number of columns */
    cuArray->init(m,
                  n);

/* Create a double pointer to a float array. It will
 * store the data from the CuArray. */
    auto NUMPY_ARRAY = new float *[1];

/* Create two double pointer int arrays that will store
 * the number rows and columns in the CuArray.
 * Btw this is what the NumPy C backend is doing every time
 * you create a numpy array in Python*/
    auto cols = new int *[1];

/* Fill the CuArray with random values */
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            cuArray->set((float) rand() / (float) RAND_MAX,
                         i,
                         j);
        }
    }

/* Copy the CuArray data into the NUMPY_ARRAY. The
 * NUMPY_ARRAY has the same dimensions as the CuArray. */
    cuArray->toNumpy(
            NUMPY_ARRAY,
            cols
    );

/* Print the NUMPY_ARRAY data and the CuArray data. */
    for (int i = 0; i < n; i++) {
        std::cout
                « cuArray->get(0,
                               i)
                « " ";
        std::cout
                « (*(NUMPY_ARRAY))[i]
                « std::endl;
    }

/* Clean this mess up. Makes you appreciate std::vectors :).*/
    delete cuArray;
    delete[] NUMPY_ARRAY[0];
    delete[] NUMPY_ARRAY;
    delete[] cols[0];
    delete[] cols;
    return 0;
}
```

Python Example

toNumpy() [2/2]

```
template<typename T >
void CuArray< T >::toNumpy (
```

```
            T ** NUMPY_ARRAY,
            int ** NUMPY_ARRAY_DIM1,
            int ** NUMPY_ARRAY_DIM2 )
```

Copies data from the CuArray to a NumPy array.

Parameters

NUMPY_ARRAY	Pointer to output NumPy array.
NUMPY_ARRAY_DIM1	Dimension 1 of the NumPy array.
NUMPY_ARRAY_DIM2	Dimension 2 of the NumPy array.

C++ Example

```
#include "cuarray.h"
#include <iostream>
#include <random>

int main() {
    std::cout
            « "Running "
            « __FILE__
            « std::endl;
/* Creates a new float CuArray instance that will have 10 rows
 * and 10 columns*/
    CuArray<float> *cuArray = new CuArray<float>();
    int m = 10; /* Number of rows */
    int n = 10; /* Number of columns */
    cuArray->init(m,
                  n);

/* Create a double pointer to a float array. It will
 * store the data from the CuArray. */
    auto NUMPY_ARRAY = new float *[1];

/* Create two double pointer int arrays that will store
 * the number rows and columns in the CuArray.
 * Btw this is what the NumPy C backend is doing every time
 * you create a numpy array in Python*/
    auto rows = new int *[1];
    auto cols = new int *[1];

/* Fill the CuArray with random values */
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            cuArray->set((float) rand() / (float) RAND_MAX,
                         i,
                         j);
        }
    }

/* Copy the CuArray data into the NUMPY_ARRAY. The
 * NUMPY_ARRAY has the same dimensions as the CuArray. */
    cuArray->toNumpy(
            NUMPY_ARRAY,
            rows,
            cols
    );

/* Print the NUMPY_ARRAY data and the CuArray data. */
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            std::cout
                    « cuArray->get(i,
                                   j)
                    « " ";
            std::cout
                    « (*(NUMPY_ARRAY))[i * m + j]
                    « std::endl;
        }
        std::cout
                « std::endl;
    }

/* Clean this mess up. Makes you appreciate std::vectors :).*/
    delete cuArray;
    delete[] NUMPY_ARRAY[0];
    delete[] NUMPY_ARRAY;
```

```
    delete[] rows[0];
    delete[] rows;
    delete[] cols[0];
    delete[] cols;
    return 0;
}
```

Python Example

```
import numpy as np

"""
Always precede CuArray with the data type
Here we are importing the CuArray float template
"""
from cuarray import FloatCuArray

print("Running", __file__)

"""Create a new float CuArray instance"""
float_cuarray = FloatCuArray()

"""
Create a random float32, 2-dimension numpy array
with 10 rows and 10 columns.
"""
np_array1 = np.random.random((10, 10)).astype(np.float32)

"""Copy the numpy array to the CuArray instance"""
float_cuarray.fromNumpy2D(np_array1)

"""Convert the CuArray instance to a numpy array"""
np_array2 = float_cuarray.toNumpy2D()

"""Print the CuArray and both numpy arrays to compare."""
for i in range(10):
    for j in range(10):
        print(
            float_cuarray[i][j],
            np_array1[i][j],
            np_array2[i][j]
        )
```

The documentation for this class was generated from the following file:

- cuarray.h

5.4 CuArrayRow< T > Class Template Reference

Public Member Functions

- **CuArrayRow** (CuArray< T > *cuArray, int i)
- T & **operator[ ]** (int i) const
- int **n** () const
- T * **data** () const

Private Attributes

- T * **data_**
- int **n_** {}

The documentation for this class was generated from the following file:

- cuarray.h

5.5 Network Class Reference

Public Member Functions

- Network ()

  Default constructor for Network.
- ∼**Network** ()

  Destructor for Network.
- void init (const std::string &trajectoryFile, const std::string &topologyFile, int firstFrame, int lastFrame, int stride=1)

  Initialize the Network with trajectory and topology files.
- int numNodes () const

  Get the number of nodes in the Network.
- CuArray< float > * nodeCoordinates ()

  Get the node coordinates as a CuArray.
- std::vector< Node * > & nodes ()

  Get a reference to the vector of nodes in the Network.
- int numFrames () const

  Get the number of frames in the Network.
- Node * nodeFromAtomIndex (int atomIndex)

  Get the node corresponding to the Atom with the given index.
- Atoms * atoms () const

  Get the Atoms object associated with the Network.
- void parsePdb (const std::string &fname)

  Parse a PDB file to populate the Network.
- void parseDcd (const std::string &nodeCoordinates, int firstFrame, int lastFrame, int stride)

  Parse a DCD file to populate the Network.
- void save (const std::string &jsonFile)

  Save the Network as a JSON file.
- void load (const std::string &jsonFile)

  Load a Network from a JSON file.
- void nodeCoordinates (const std::string &nodeCoordinatesFile)

  Set the node coordinates from a file.

Private Attributes

- std::vector< Node * > **nodeAtomIndexVector_**
- std::vector< Node * > **nodes_**
- int **numNodes_**
- int **numFrames_**
- CuArray< float > * **nodeCoordinates_**
- Atoms * **atoms_**

5.5.1 Constructor & Destructor Documentation

Network()

```
Network::Network ( )
```

Default constructor for Network.

Constructs an empty Network object.

5.5.2 Member Function Documentation

atoms()

```
Atoms * Network::atoms ( ) const
```

Get the Atoms object associated with the Network.

Returns a pointer to the Atoms object associated with the Network.

Returns

A pointer to the Atoms object.

init()

```
void Network::init (
            const std::string & trajectoryFile,
            const std::string & topologyFile,
            int firstFrame,
            int lastFrame,
            int stride = 1 )
```

Initialize the Network with trajectory and topology files.

Initializes the Network by loading trajectory and topology files.

Parameters

trajectoryFile	Path to the trajectory file.
topologyFile	Path to the topology file.
firstFrame	Index of the first frame to consider.
lastFrame	Index of the last frame to consider.
stride	Stride between frames.

load()

```
void Network::load (
            const std::string & jsonFile )
```

Load a Network from a JSON file.

Loads a Network from the specified JSON file.

Parameters

jsonFile	Path to the JSON file.

nodeCoordinates() [1/2]

```
CuArray< float > * Network::nodeCoordinates ( )
```

Get the node coordinates as a CuArray.

Returns a pointer to the CuArray object containing the node coordinates.

Returns

A pointer to the CuArray containing the node coordinates.

nodeCoordinates() [2/2]

```
void Network::nodeCoordinates (
            const std::string & nodeCoordinatesFile )
```

Set the node coordinates from a file.

Sets the node coordinates from the specified node coordinates file.

Parameters

nodeCoordinatesFile	Path to the node coordinates file.

nodeFromAtomIndex()

```
Node * Network::nodeFromAtomIndex (
            int atomIndex )
```

Get the node corresponding to the Atom with the given index.

Returns a pointer to the Node object that the Atom with the specified index is part of

Parameters

atomIndex	The index of the Atom.

Returns

A pointer to the Node corresponding to the Atom index.

nodes()

```
std::vector< Node * > & Network::nodes ( )
```

Get a reference to the vector of nodes in the Network.

Returns a reference to the vector of nodes in the Network.

Returns

A reference to the vector of nodes.

numFrames()

```
int Network::numFrames ( ) const
```

Get the number of frames in the Network.

Returns the number of frames in the Network.

Returns

The number of frames.

numNodes()

```
int Network::numNodes ( ) const
```

Get the number of nodes in the Network.

Returns the number of nodes in the Network.

Returns

The number of nodes.

parseDcd()

```
void Network::parseDcd (
            const std::string & nodeCoordinates,
            int firstFrame,
            int lastFrame,
            int stride )
```

Parse a DCD file to populate the Network.

Parses the specified DCD file to populate the Network with node coordinates.

Parameters

fname	Path to the node coordinates file.
firstFrame	Index of the first frame to consider.
lastFrame	Index of the last frame to consider.
stride	Stride between frames.

parsePdb()

```
void Network::parsePdb (
            const std::string & fname )
```

Parse a PDB file to populate the Network.

Parses the specified PDB file to populate the Network with Atom and Node objects.

Parameters

fname	Path to the PDB file.

save()

```
void Network::save (
            const std::string & jsonFile )
```

Save the Network as a JSON file.

Saves the Network as a JSON file.

Parameters

jsonFile	Path to the JSON file.

The documentation for this class was generated from the following file:

- network.h

5.6 Node Class Reference

Represents a node in a graph.

`#include <node.h>`

Public Member Functions

- **Node** ()
  Default constructor for Node.
- ∼**Node** ()
  Destructor for Node.
- Node (unsigned int numFrames, unsigned int index_)
  Constructor for Node with specified number of frames and index.
- void addAtom (Atom ∗atom, CuArray< float > ∗coordinates, CuArray< float > ∗nodeCoordinates)
  Add an Atom to the Node.
- std::string tag ()
  Get the tag of the Node.

- unsigned int numAtoms () const

  Get the number of Atoms in the Node.
- unsigned int index () const

  Get the index of the Node.
- float totalMass () const

  Get the total mass of the Node.
- unsigned int hash () const

  Get the hash value of the Node.
- std::vector< Atom * > atoms () const

  Get a vector of pointers to the Atoms in the Node.

Private Attributes

- unsigned int **_numAtoms**
- std::vector< int > **atomIndices_**
- unsigned int **_index**
- float **_totalMass**
- std::string **_tag**
- std::vector< Atom * > **atoms_**
- unsigned int **_hash** = 0
- unsigned int **_numFrames**

Friends

- class **Network**

5.6.1 Detailed Description

Represents a node in a graph.

5.6.2 Constructor & Destructor Documentation

Node()

```
Node::Node (
            unsigned int numFrames,
            unsigned int index_ )
```

Constructor for Node with specified number of frames and index.

Parameters

numFrames	Number of frames.
*index_*	Index of the node.

5.6.3 Member Function Documentation

addAtom()

```
void Node::addAtom (
            Atom * atom,
            CuArray< float > * coordinates,
            CuArray< float > * nodeCoordinates )
```

Add an Atom to the Node.

Adds the specified Atom to the Node along with its corresponding coordinates.

Parameters

atom	Pointer to the Atom object.
coordinates	Pointer to the coordinates array.
nodeCoordinates	Pointer to the node coordinates array.

atoms()

```
std::vector< Atom * > Node::atoms ( ) const
```

Get a vector of pointers to the Atoms in the Node.

Returns a vector of pointers to the Atoms contained in the Node.

Returns

A vector of pointers to the Atoms in the Node.

hash()

```
unsigned int Node::hash ( ) const
```

Get the hash value of the Node.

Returns the hash value of the Node, which is a unique identifier based on its tag and index.

Returns

The hash value of the Node.

index()

```
unsigned int Node::index ( ) const
```

Get the index of the Node.

Returns the index of the Node.

Returns

The index of the Node.

numAtoms()

```
unsigned int Node::numAtoms ( ) const
```

Get the number of Atoms in the Node.

Returns the number of Atoms contained in the Node.

Returns

The number of Atoms in the Node.

tag()

```
std::string Node::tag ( )
```

Get the tag of the Node.

Returns the tag of the Node, which represents its unique identifier.

Returns

The tag of the Node.

totalMass()

```
float Node::totalMass ( ) const
```

Get the total mass of the Node.

Returns the total mass of the Node, calculated as the sum of the masses of all the Atoms in the Node.

Returns

The total mass of the Node.

The documentation for this class was generated from the following file:

- node.h

Index

addAtom
  Atoms, 16
  Node, 61
allocatedDevice
  CuArray< T >, 20
allocateDevice
  CuArray< T >, 21
allocatedHost
  CuArray< T >, 22
allocateHost
  CuArray< T >, 22
argsort
  CuArray< T >, 23
at
  Atoms, 16
Atom, 8
  Atom, 9, 10
  chainId, 10
  element, 10
  hash, 11
  index, 11
  load, 11
  mass, 12
  name, 12
  occupancy, 12
  residueId, 12
  residueName, 12
  segmentId, 13
  serial, 13
  tag, 13
  temperatureFactor, 13
  x, 14
  y, 14
  z, 15
Atoms, 15
  addAtom, 16
  at, 16
  Atoms, 16
  atoms, 16
  numAtoms, 17
  numUniqueTags, 17
atoms
  Atoms, 16
  Network, 56
  Node, 61

bytes
  CuArray< T >, 25

chainId
  Atom, 10
CuArray
  CuArray< T >, 20
CuArray< T >, 17
  allocatedDevice, 20
  allocateDevice, 21
  allocatedHost, 22
  allocateHost, 22
  argsort, 23
  bytes, 25
  CuArray, 20
  deallocateDevice, 26
  deallocateHost, 26
  device, 27
  fromCuArrayDeepCopy, 27
  fromCuArrayShallowCopy, 29
  fromNumpy, 31, 32
  get, 33
  host, 35
  init, 35, 36
  load, 38
  m, 40
  n, 40
  operator[], 41
  owner, 42
  save, 44
  set, 45
  size, 48
  sort, 48
  toDevice, 50
  toHost, 50
  toNumpy, 51, 52
CuArrayRow< T >, 54

deallocateDevice
  CuArray< T >, 26
deallocateHost
  CuArray< T >, 26
device
  CuArray< T >, 27

element
  Atom, 10

fromCuArrayDeepCopy
  CuArray< T >, 27
fromCuArrayShallowCopy
  CuArray< T >, 29
fromNumpy
  CuArray< T >, 31, 32

generalizedCorrelation
  netcalc, 4
generalizedCorrelationCpu
  netcalc, 4
generalizedCorrelationGpu
  netcalc, 5
generateRestartAbFromCheckpointFile
  netcalc, 5
get
  CuArray< T >, 33

hash

Atom, 11
Node, 61
host
CuArray< T >, 35

index
Atom, 11
Node, 61
init
CuArray< T >, 35, 36
Network, 56

load
Atom, 11
CuArray< T >, 38
Network, 56

m
CuArray< T >, 40
mass
Atom, 12
mutualInformation
netcalc, 6
mutualInformationCpu
netcalc, 7
mutualInformationGpu
netcalc, 8

n
CuArray< T >, 40
name
Atom, 12
netcalc, 3
generalizedCorrelation, 4
generalizedCorrelationCpu, 4
generalizedCorrelationGpu, 5
generateRestartAbFromCheckpointFile, 5
mutualInformation, 6
mutualInformationCpu, 7
mutualInformationGpu, 8
NetSci: A Toolkit for High Performance Scientific Network Analysis Computation, 1
Network, 55
atoms, 56
init, 56
load, 56
Network, 55
nodeCoordinates, 56, 57
nodeFromAtomIndex, 57
nodes, 57
numFrames, 58
numNodes, 58
parseDcd, 58
parsePdb, 58
save, 59
Node, 59
addAtom, 61
atoms, 61
hash, 61
index, 61
Node, 60
numAtoms, 61
tag, 62
totalMass, 62
nodeCoordinates
Network, 56, 57
nodeFromAtomIndex
Network, 57
nodes
Network, 57
numAtoms
Atoms, 17
Node, 61
numFrames
Network, 58
numNodes
Network, 58
numUniqueTags
Atoms, 17

occupancy
Atom, 12
operator[]
CuArray< T >, 41
owner
CuArray< T >, 42

parseDcd
Network, 58
parsePdb
Network, 58

residueId
Atom, 12
residueName
Atom, 12

save
CuArray< T >, 44
Network, 59
segmentId
Atom, 13
serial
Atom, 13
set
CuArray< T >, 45
size
CuArray< T >, 48
sort
CuArray< T >, 48

tag
Atom, 13
Node, 62
temperatureFactor
Atom, 13
toDevice
CuArray< T >, 50

toHost
CuArray< T >, 50
toNumpy
CuArray< T >, 51, 52
totalMass
Node, 62

x
Atom, 14

y
Atom, 14

z
Atom, 15