Report 2060042-TN-01 D1.1: Implementation of TensorRegions library in Nektar++

Chris Cantwell & Spencer Sherwin, Imperial College London,
David Moxey, King's College London

30th March 2022

Contents

1 Introduction

This report outlines the initial stages of development of the TensorRegions library in line with Deliverable 1.1. Nektar++ currently supports high-order finite element expansions on one-dimensional, two-dimensional and three-dimensional elements, including a range of regular and simplex elements. The need to solve higher-dimensional partial differential equations (PDEs) in modelling plasma kinetics – such as the Vlasov-Poisson equation, necessitates an extension of the code to support higher-dimensional finite element spaces.

The construction of these high-dimensional problems naturally lends itself to a formulation of the corresponding high-dimensional finite element space as a tensor product of lower-dimensional finite element spaces. This has several advantages, obtained from the ability to leverage existing infrastructure in Nektar++, including basis functions, finite element operators and solver techniques. The TensorRegions library aims to capitalise on this concept and provide a framework in which high-dimensional PDEs can be solved.

2 Implementation

This deliverable comprises an implementation of the underpinning objects with documentation describing their API and use (appended to this document). This lays the foundation for deliverable 2.1, which provides implementation of the core finite element operators within the TensorRegions library.

The code is available in MR 1325 of the Nektar++ project:

`https://gitlab.nektar.info/nektar/nektar/-/merge_requests/1325`

The TensorRegions library is currently comprised of three core classes:

- `TensorRegion`: This encapsulates a high-dimensional finite element space, constructed as a tensor-product of two or more lower-dimensional finite element spaces.
- `TensorStorage`: This encapsulates a high-dimensional solution on a `TensorRegion`, allowing access to the data also through lower-dimensional *slices* on which the existing lower-dimensional algorithms and infrastructure can be applied.
- `TensorStorage::View`: This provides transparent access to a lower-dimensional subspace of the solution.

2.1 The `TensorRegion` class

Construction of a `TensorRegion` is from the constituent `MultiRegions` objects. These should be constructed a priori and passed to the `TensorRegion`.

Member routines are provided to query the number of dimensions and the size of those respective dimensions. Placeholders for finite element operators are required, which will be completed for Deliverable 2.1.

Storage for a solution to a high-dimensional problem on a `TensorRegion` is not stored internally, but rather in the separate `TensorStorage` class (see Section 2.2.

2.2 The `TensorStorage` class

The `TensorStorage` class is templated on the type of data to be stored. This is commonly `NekDouble` (aka `double`) in Nektar++, but enables future support for mixed-precision.

`TensorStorage` objects should be initialised through an existing `TensorRegion` object. This ensures it is configured with the correct dimensions compatible with the underlying finite element expansions, but also passes a pointer to the `TensorRegion` object into the `TensorStorage` object to allow for later interrogation.

Member routines are provided to query the number of dimensions and the size of those respective dimensions. The `[]` operator is overloaded to allow linear access to the data. Data is ordered with the first dimension indexed fastest.

The `TensorStorage` object is also aware of the nature of the data being stored. In the context of Nektar++ this indicates whether the data represents physical space values, or spectral/hp element coefficients.

Finally, the class enables the construction of lower-dimensional views on the data.

2.2.1 The `TensorStorage::View` class

The `TensorStorage::View` class provides a view onto a low-dimensional slice through a `TensorStorage` object. It allows both read and write access to the underlying data.

3 Example Usage

In this section, we outline the example usage of the currently available data structures. This code is taken from the demo code available in:

`nektar++/library/Demos/TensorRegions/Domain.cpp`

To use the TensorRegions library, we first need to include the necessary headers from Nektar++. As well as `TensorRegion.h`, we also need to include headers for the constituent `MultiRegion` classes and the session reader.

```
#include <LibUtilities/BasicUtils/SessionReader.h>
#include <SpatialDomains/MeshGraph.h>
#include <MultiRegions/DisContField.h>
#include <TensorRegions/TensorRegion.h>
```

3.1 Constructing the constituent `MultiRegions` objects

At the start of the program, we need to establish a session and read in the mesh. The input session should comprise of two or more domains, which will be used to initialise the different `MultiRegions` objects on which the TensorRegion is based.

```
LibUtilities::SessionReaderSharedPtr vSession =
            LibUtilities::SessionReader::CreateInstance(argc, argv);

// read in mesh
SpatialDomains::MeshGraphSharedPtr graph1D =
            SpatialDomains::MeshGraph::Read(vSession);
```

We can now identify the number of domains present and extract information about the composites and boundary conditions used:

```
// Read the geometry and the expansion information
size_t nDomains = graph1D->GetDomain().size();
const std::vector<SpatialDomains::CompositeMap> domain = graph1D->GetDomain();
SpatialDomains::BoundaryConditions Allbcs(vSession, graph1D);
bool SetToOneSpaceDimension = true;
```

We are now ready to construct the low-dimensional constituent expansion objects on which our TensorRegion will be based:

```
// Construct the constituent expansions
std::vector<MultiRegions::ExpListSharedPtr> exps(nDomains);
for (unsigned int i = 0; i < nDomains; ++i)
{
    exps[i] = MemoryManager<MultiRegions::DisContField>::AllocateSharedPtr(
                vSession, graph1D, domain[i], Allbcs,
                vSession->GetVariable(0), SetToOneSpaceDimension);
}
```

3.2 Initialising a TensorRegion

A `TensorRegion` object maybe initialised explicitly from two MultiRegions objects, or from a `std::vector` of MultiRegion objects. We use the latter in this case, even though the example only contains two domains in the tensor.

```
// Construct a TensorRegion from the expansions
TensorRegions::TensorRegion tr(exps);
```

3.3 Initialising storage

To store a solution on a `TensorRegion`, we utilise the `TensorStorage` class. To allocate such an object of appropriate size, we do not instantiate it directly, but rather call the member function `AllocateStorage()` of the `TensorRegion` class, which correctly configures the storage object for that `TensorRegion`. We can print out associated characteristics of the `TensorStorage` object to verify it has been created correctly.

```
// Allocate a TensorStorage object to hold a discrete solution
TensorRegions::TensorRegionStorageSharedPtr st = tr.AllocateStorage();
cout << "Number of dimensions:      " << st->dims() << endl;
cout << "Total tensor storage size: " << st->size() << endl;
for (size_t i = 0; i < st->dims(); ++i)
{
    cout << "Dim " << i << " has size " << st->size(i) << " points." << endl;
}
```

3.4 Views on a TensorStorage object

A `TensorStorage::View` object presents a read-write interface to a subset of the underlying tensor storage object. This enables transparent operation along a slice of the storage. For performance reasons, one can extract this slice into a separate (and contiguous in memory) Nektar++ `Array` object. This data can also be injected back into the underlying `TensorStorage` object (not shown).

```
// Construct views on the TensorStorage object to operate on slices of the
// data along different directions.
TensorRegions::TensorRegionStorage::View st0 = st->slice(0, {0,0});
TensorRegions::TensorRegionStorage::View st1 = st->slice(1, {0,0});
cout << "Size of slice 0 is " << st0.size() << endl;
cout << "Size of slice 1 is " << st1.size() << endl;

// Update TensorStorage object through View.
for (int i = 0; i < st1.size(); ++i) {
    st1[i] = i;
}

// Extract Array data from View
Array<OneD, NekDouble> data0 = st0.extract();
Array<OneD, NekDouble> data1 = st1.extract();
cout << "Size of slice 0 data is " << data0.size() << endl;
cout << "Size of slice 1 data is " << data1.size() << endl;

// Display all entries of TensorStorage object.
for (int i = 0; i < st->size(); ++i) {
    cout << (*st)[i] << endl;
}
```

4 API Documentation

The remainder of this report documents the programming API to the TensorRegions library.

Nektar++ TensorRegions

Generated by Doxygen 1.9.1

Chapter 1

Namespace Index

1.1 Namespace List

Here is a list of all namespaces with brief descriptions:

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 3

Class Index

3.1 Class List

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

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

Chapter 5

Namespace Documentation

5.1 Nektar Namespace Reference

Namespaces

- TensorRegions

5.2 Nektar::TensorRegions Namespace Reference

Classes

- class TensorRegion

  High-dimensional region, comprising of a tensor-product of two MultiRegions.
- class TensorStorage

  Storage container for TensorRegions solution.

Typedefs

- using TensorRegionStorage = TensorStorage< NekDouble >

  Double-precision floating-point specialisation of a TensorStorage.
- using TensorRegionStorageSharedPtr = std::shared_ptr< TensorRegionStorage >

  Shared pointer to a double-precision floating-point TensorStorage object.
- using TensorRegionSharedPtr = std::shared_ptr< TensorRegion >

  A shared pointer to a TensorRegion object.
- template< typename DataType >
  using TensorStorageSharedPtr = std::shared_ptr< TensorStorage< DataType > >

  A shared pointer to a TensorStorage object.

5.2.1 Typedef Documentation

5.2.1.1 TensorRegionSharedPtr

```
using Nektar::TensorRegions::TensorRegionSharedPtr = typedef std::shared_ptr<TensorRegion>
```

A shared pointer to a TensorRegion object.

Definition at line 67 of file TensorRegion.h.

5.2.1.2 TensorRegionStorage

```
using Nektar::TensorRegions::TensorRegionStorage = typedef TensorStorage<NekDouble>
```

Double-precision floating-point specialisation of a TensorStorage.

Definition at line 15 of file TensorRegion.h.

5.2.1.3 TensorRegionStorageSharedPtr

```
using Nektar::TensorRegions::TensorRegionStorageSharedPtr = typedef std::shared_ptr<TensorRegionStorage>
```

Shared pointer to a double-precision floating-point TensorStorage object.

Definition at line 17 of file TensorRegion.h.

5.2.1.4 TensorStorageSharedPtr

```
template<typename DataType >
using Nektar::TensorRegions::TensorStorageSharedPtr = typedef std::shared_ptr<TensorStorage<Data↩
Type> >
```

A shared pointer to a TensorStorage object.

Definition at line 376 of file TensorStorage.hpp.

5.3 TensorRegions Namespace Reference

Namespace encapsulating algorithms and data structures for constructing high-dimensional high-order finite element spaces.

5.3.1 Detailed Description

Namespace encapsulating algorithms and data structures for constructing high-dimensional high-order finite element spaces.

Chapter 6

Class Documentation

6.1 Nektar::TensorRegions::TensorRegion Class Reference

High-dimensional region, comprising of a tensor-product of two MultiRegions.

```
#include <TensorRegion.h>
```

Public Member Functions

- TensorRegion (std::vector< MultiRegions::ExpListSharedPtr > exp)

  Construct a new TensorRegion object from one or more MultiRegions objects.
- TensorRegion (const TensorRegion &pSrc)

  Construct a new TensorRegion object from an existing object.
- ∼TensorRegion ()

  Destroy the TensorRegion object.
- TensorRegionStorageSharedPtr AllocateStorage ()

  Allocates a compatible TensorStorage object.
- MultiRegions::ExpListSharedPtr GetExpList (int index)

  Access a constituent MultiRegions object.
- size_t GetNumPoints ()
- size_t GetNumCoeffs ()
- void BwdTrans (const TensorStorage< NekDouble > &inarray, TensorStorage< NekDouble > &outarray)
- void IProductWRTBase (const TensorStorage< NekDouble > &inarray, TensorStorage< NekDouble > &outarray)
- void IProductWRTDerivBase (unsigned int dir, const TensorStorage< NekDouble > &inarray, TensorStorage< NekDouble > &outarray)
- void PhysDeriv (unsigned int dir, const TensorStorage< NekDouble > &inarray, TensorStorage< NekDouble > &outarray)
- void FwdTrans (const TensorStorage< NekDouble > &inarray, TensorStorage< NekDouble > &outarray)
- void GetCoords (Array< OneD, Array< OneD, NekDouble >> coords)
- NekDouble PhysEvaluate (Array< OneD, NekDouble > coords)

Private Attributes

- std::vector< MultiRegions::ExpListSharedPtr > m_exp

  List of constituent expansion lists.
- size_t m_numPoints

  Total number of points in tensor.
- size_t m_numCoeffs

  Total number of coefficients in tensor.

6.1.1 Detailed Description

High-dimensional region, comprising of a tensor-product of two MultiRegions.

A TensorRegion represents a high-dimensional finite element expansion. It is constructed through a tensor-product of two or more lower- dimensional MultiRegions expansions, defined by independent meshes. The solution storage for a TensorRegion is provided by the TensorStorage class. Finite element operators are constructed by applying the sub-factorisation approach, deferring to the lower-dimensional consistuent MultiRegions classes for their implementation.

Definition at line 20 of file TensorRegion.h.

6.1.2 Constructor & Destructor Documentation

6.1.2.1 TensorRegion() `[1/2]`

```
Nektar::TensorRegions::TensorRegion::TensorRegion (
            std::vector< MultiRegions::ExpListSharedPtr > subspaces )
```

Construct a new TensorRegion object from one or more MultiRegions objects.

Parameters

subspaces	Vector of constituent subspaces.

Definition at line 27 of file TensorRegion.cpp.

```
{
    ASSERTL0(subspaces.size() <= 2, "TensorRegion currently only supports 2 subspaces.")
    m_exp = subspaces;
    m_numPoints = 1;
    m_numCoeffs = 1;
    for (auto &e : m_exp)
    {
        m_numPoints *= e->GetNpoints();
        m_numCoeffs *= e->GetNcoeffs();
    }
}
```

References m_exp, m_numCoeffs, and m_numPoints.

6.1.2.2 TensorRegion() [2/2]

```
Nektar::TensorRegions::TensorRegion::TensorRegion (
            const TensorRegion & pSrc )
```

Construct a new TensorRegion object from an existing object.

Parameters

pSrc	Existing TensorRegion object.

Definition at line 45 of file TensorRegion.cpp.

```
    : m_exp(pSrc.m_exp),
      m_numPoints(pSrc.m_numPoints),
      m_numCoeffs(pSrc.m_numCoeffs)
{
    // Nothing to do
}
```

6.1.2.3 ∼TensorRegion()

```
Nektar::TensorRegions::TensorRegion::∼TensorRegion ( )
```

Destroy the TensorRegion object.

Definition at line 57 of file TensorRegion.cpp.

```
{

}
```

6.1.3 Member Function Documentation

6.1.3.1 AllocateStorage()

```
TensorRegionStorageSharedPtr Nektar::TensorRegions::TensorRegion::AllocateStorage ( )
```

Allocates a compatible TensorStorage object.

Returns

TensorRegionStorageSharedPtr

Definition at line 67 of file TensorRegion.cpp.

```
{
    std::vector<size_t> sizes;
    for (int i = 0; i < m_exp.size(); ++i)
    {
        sizes.push_back(m_exp[i]->GetNpoints());
    }
    return TensorRegionStorage::create(sizes);
}
```

References Nektar::TensorRegions::TensorStorage< DataType >::create(), and m_exp.

6.1.3.2 BwdTrans()

```
void Nektar::TensorRegions::TensorRegion::BwdTrans (
            const TensorStorage< NekDouble > & inarray,
            TensorStorage< NekDouble > & outarray )
```

Definition at line 101 of file TensorRegion.cpp.

```
{
    boost::ignore_unused(inarray, outarray);

    // const int nm0 = m_exp[0]->GetNcoeffs();
    // const int nm1 = m_exp[1]->GetNcoeffs();
    // const int np0 = m_exp[0]->GetNpoints();
    // const int np1 = m_exp[1]->GetNpoints();

    // ASSERTL0(inarray.size() >= nm0*nm1, "Input array is of incorrect size.");
    // ASSERTL0(outarray.size() >= np0*np1, "Output array is of incorrect size.");

    // Array<OneD, NekDouble> x;

    // // Apply first dimension bwdtrans
    // for (int i = 0; i < nm1; ++i) {
    //     m_exp[0]->BwdTrans(inarray + i*nm0, x = outarray + i*np0);
    // }

    // // Transpose data
    // Array<OneD, NekDouble> tmp(np0*nm1);
    // for (int j = 0; j < nm1; ++j) {
    //     for (int i = 0; i < np0; ++i) {
    //         tmp[i*nm1 + j] = outarray[j*np0 + i];
    //     }
    // }

    // // Apply second dimension bwdtrans
    // for (int j = 0; j < np0; ++j) {
    //     m_exp[1]->BwdTrans(tmp + j*np0, x = outarray + j*np1);
    // }
}
```

6.1.3.3 FwdTrans()

```
void Nektar::TensorRegions::TensorRegion::FwdTrans (
            const TensorStorage< NekDouble > & inarray,
            TensorStorage< NekDouble > & outarray )
```

Definition at line 159 of file TensorRegion.cpp.

```
{
    boost::ignore_unused(inarray, outarray);
}
```

6.1.3.4 GetCoords()

```
void Nektar::TensorRegions::TensorRegion::GetCoords (
            Array< OneD, Array< OneD, NekDouble >> coords )
```

6.1.3.5 GetExpList()

```
MultiRegions::ExpListSharedPtr Nektar::TensorRegions::TensorRegion::GetExpList (
            int index )
```

Access a constituent MultiRegions object.

Parameters

index	The index of the MultiRegion object to access.

Returns

MultiRegions::ExpListSharedPtr

Definition at line 83 of file TensorRegion.cpp.

```
{
    ASSERTL0(index >= m_exp.size(), "Index out of range.");

    return m_exp[index];
}
```

References m_exp.

6.1.3.6 GetNumCoeffs()

```
size_t Nektar::TensorRegions::TensorRegion::GetNumCoeffs ( )
```

Definition at line 96 of file TensorRegion.cpp.

```
{
    return m_numCoeffs;
}
```

References m_numCoeffs.

6.1.3.7 GetNumPoints()

```
size_t Nektar::TensorRegions::TensorRegion::GetNumPoints ( )
```

Definition at line 91 of file TensorRegion.cpp.

```
{
    return m_numPoints;
}
```

References m_numPoints.

6.1.3.8 IProductWRTBase()

```
void Nektar::TensorRegions::TensorRegion::IProductWRTBase (
             const TensorStorage< NekDouble > & inarray,
             TensorStorage< NekDouble > & outarray )
```

Definition at line 136 of file TensorRegion.cpp.

```
{
    boost::ignore_unused(inarray, outarray);
}
```

6.1.3.9 IProductWRTDerivBase()

```
void Nektar::TensorRegions::TensorRegion::IProductWRTDerivBase (
            unsigned int dir,
            const TensorStorage< NekDouble > & inarray,
            TensorStorage< NekDouble > & outarray )
```

Definition at line 143 of file TensorRegion.cpp.

```
{
    boost::ignore_unused(dir, inarray, outarray);
}
```

6.1.3.10 PhysDeriv()

```
void Nektar::TensorRegions::TensorRegion::PhysDeriv (
            unsigned int dir,
            const TensorStorage< NekDouble > & inarray,
            TensorStorage< NekDouble > & outarray )
```

Definition at line 151 of file TensorRegion.cpp.

```
{
    boost::ignore_unused(dir, inarray, outarray);
}
```

6.1.3.11 PhysEvaluate()

```
NekDouble Nektar::TensorRegions::TensorRegion::PhysEvaluate (
            Array< OneD, NekDouble > coords )
```

6.1.4 Member Data Documentation

6.1.4.1 m_exp

```
std::vector<MultiRegions::ExpListSharedPtr> Nektar::TensorRegions::TensorRegion::m_exp  [private]
```

List of constituent expansion lists.

Definition at line 61 of file TensorRegion.h.

Referenced by AllocateStorage(), GetExpList(), and TensorRegion().

6.1.4.2 m_numCoeffs

```
size_t Nektar::TensorRegions::TensorRegion::m_numCoeffs  [private]
```

Total number of coefficients in tensor.

Definition at line 63 of file TensorRegion.h.

Referenced by GetNumCoeffs(), and TensorRegion().

6.1.4.3 m_numPoints

```
size_t Nektar::TensorRegions::TensorRegion::m_numPoints  [private]
```

Total number of points in tensor.

Definition at line 62 of file TensorRegion.h.

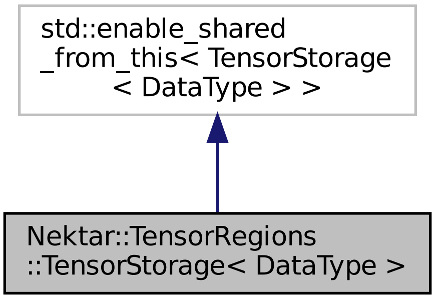
Referenced by GetNumPoints(), and TensorRegion().

6.2 Nektar::TensorRegions::TensorStorage< DataType > Class Template Reference

Storage container for TensorRegions solution.

```
#include <TensorStorage.hpp>
```

Inheritance diagram for Nektar::TensorRegions::TensorStorage< DataType >:



Classes

- class View

  Represents a one-dimensional view onto a TensorStorage object.

Public Types

- enum State { ePhys , eCoeff }

  Describes whether storage represents physical values or spectral/hp element coefficients.

Public Member Functions

- TensorStorage (const TensorStorage &pSrc)

  Construct a new TensorStorage object from an existing object.
- TensorStorage (TensorStorage &&pSrc)

  Construct a new TensorStorage object from an existing object (move semantics).
- virtual ∼TensorStorage ()

  Destroy a TensorRegions object.
- size_t dims ()

  Returns the number of dimensions of the TensorStorage object.
- size_t size ()

  Returns the total size of the TensorStorage object.
- size_t size (const size_t idx)

  Returns the size of the TensorStorage object.
- DataType & operator[ ] (size_t idx)

  Access an element of the storage container.
- DataType operator[ ] (size_t idx) const

  Access an element of the storage container.
- enum State getState ()

  Returns whether the storage represents physical solution or spectral/hp element coefficients.
- void setState (enum State state=ePhys)

  Sets whether storage is in physical space or coefficient space.
- View slice (unsigned int dim, const std::vector< size_t > &coord)

  Constructs a TensorStorage:::View on a high-dimensional TensorStorage object, corresponding to a single constituent dimension.

Static Public Member Functions

- static std::shared_ptr< TensorStorage< DataType > > create (const size_t size1, const size_t size2, const DataType val=0.0)

  Factory for creating a TensorStorage object.
- static std::shared_ptr< TensorStorage< DataType > > create (const std::vector< size_t > sizes, const DataType val=0.0)

  Factory for creating a TensorStorage object.

Private Member Functions

- TensorStorage (const size_t size1, const size_t size2, const DataType val=0.0)

  Construct a new TensorStorage object with two given sizes.
- TensorStorage (const std::vector< size_t > sizes, const DataType val=0.0)

  Construct a new TensorStorage object with two given sizes.

Private Attributes

- Array< OneD, NekDouble > m_data

  Storage for actual data.
- std::vector< size_t > m_sizes

  Sizes for each sub-component of the tensor.
- enum TensorStorage::State m_state = ePhys

  Flag indicating if data is in coeff or physical state.

6.2.1 Detailed Description

template<typename DataType>
class Nektar::TensorRegions::TensorStorage< DataType >

Storage container for TensorRegions solution.

Data is stored fastest in the lowest dimension.

Template Parameters

DataType	The type of data stored by the container.

Definition at line 28 of file TensorStorage.hpp.

6.2.2 Member Enumeration Documentation

6.2.2.1 State

```
template<typename DataType >
enum Nektar::TensorRegions::TensorStorage::State
```

Describes whether storage represents physical values or spectral/hp element coefficients.

Enumerator

ePhys	Physical values.
eCoeff	Spectral/hp element coefficients.

Definition at line 36 of file TensorStorage.hpp.

```
36                  {
37              ePhys,      ///< Physical values
38              eCoeff      ///< Spectral/hp element coefficients
39          };
```

6.2.3 Constructor & Destructor Documentation

6.2.3.1 TensorStorage() [1/4]

```
template<typename DataType >
Nektar::TensorRegions::TensorStorage< DataType >::TensorStorage (
            const TensorStorage< DataType > & pSrc ) [inline]
```

Construct a new TensorStorage object from an existing object.

Parameters

pSrc	Existing TensorStorage object.

Definition at line 208 of file TensorStorage.hpp.

```
            : m_data(pSrc.m_data),
              m_sizes(pSrc.m_sizes)
        {
            // Nothing to do
        }
```

6.2.3.2 TensorStorage() [2/4]

```
template<typename DataType >
Nektar::TensorRegions::TensorStorage< DataType >::TensorStorage (
            TensorStorage< DataType > && pSrc ) [inline]
```

Construct a new TensorStorage object from an existing object (move semantics).

Parameters

pSrc	Existing temporary TensorStorage object.

Definition at line 221 of file TensorStorage.hpp.

```
            : Array<OneD, DataType>(std::move(pSrc)),
              m_sizes(std::move(pSrc.m_sizes))
        {
            // Nothing to do
        }
```

6.2.3.3 ∼TensorStorage()

```
template<typename DataType >
virtual Nektar::TensorRegions::TensorStorage< DataType >::∼TensorStorage ( ) [inline], [virtual]
```

Destroy a TensorRegions object.

Definition at line 232 of file TensorStorage.hpp.

```
        {

        }
```

6.2.3.4 TensorStorage() [3/4]

```
template<typename DataType >
Nektar::TensorRegions::TensorStorage< DataType >::TensorStorage (
              const size_t size1,
              const size_t size2,
              const DataType val = 0.0 )  [inline], [private]
```

Construct a new TensorStorage object with two given sizes.

Parameters

size1	First component size.
size2	Second component size.
val	Initial value for all elements.

Definition at line 348 of file TensorStorage.hpp.

```
            : m_data(Array<OneD, DataType>(size1 * size2, val)),
              m_sizes({size1, size2})
        {
            // Nothing to do
        }
```

6.2.3.5 TensorStorage() [4/4]

```
template<typename DataType >
Nektar::TensorRegions::TensorStorage< DataType >::TensorStorage (
              const std::vector< size_t > sizes,
              const DataType val = 0.0 )  [inline], [private]
```

Construct a new TensorStorage object with two given sizes.

Parameters

sizes	Vector of arbitrary number of sizes > 0
val	Initial vlaue for all elements.

Definition at line 361 of file TensorStorage.hpp.

```
            : m_sizes(sizes)
        {
            ASSERTL1(!sizes.empty(), "Must have at least one size.");
            int size = 1;
            for (auto &s : sizes)
            {
                size *= s;
            }
            m_data = Array<OneD, NekDouble>(size, val);
        }
```

References Nektar::TensorRegions::TensorStorage< DataType >::m_data, and Nektar::TensorRegions::TensorStorage< DataType >::size().

6.2.4 Member Function Documentation

6.2.4.1 create() [1/2]

```
template<typename DataType >
static std::shared_ptr<TensorStorage<DataType> > Nektar::TensorRegions::TensorStorage< DataType >::create (
            const size_t size1,
            const size_t size2,
            const DataType val = 0.0 ) [inline], [static]
```

Factory for creating a TensorStorage object.

Parameters

size1	Size in first dimension.
size2	Size in second dimension.
val	Default value.

Returns

std::shared_ptr<TensorStorage<DataType>>

Definition at line 183 of file TensorStorage.hpp.

```
        {
            ASSERTL0(size1 > 0, "Size1 must be greater than zero.");
            ASSERTL0(size2 > 0, "Size2 must be greater than zero.");

            return std::shared_ptr<TensorStorage<DataType»(new TensorStorage<DataType>(size1, size2, val));
        }
```

Referenced by Nektar::TensorRegions::TensorRegion::AllocateStorage().

6.2.4.2 create() [2/2]

```
template<typename DataType >
static std::shared_ptr<TensorStorage<DataType> > Nektar::TensorRegions::TensorStorage< DataType >::create (
            const std::vector< size_t > sizes,
            const DataType val = 0.0 ) [inline], [static]
```

Factory for creating a TensorStorage object.

Parameters

sizes	Vector of sizes for all dimensions.
val	Default value.

Returns

std::shared_ptr<TensorStorage<DataType>>

Definition at line 198 of file TensorStorage.hpp.

```
{
    return std::shared_ptr<TensorStorage<DataType»(new TensorStorage<DataType>(sizes, val));
}
```

6.2.4.3 dims()

```
template<typename DataType >
size_t Nektar::TensorRegions::TensorStorage< DataType >::dims ( )  [inline]
```

Returns the number of dimensions of the TensorStorage object.

Definition at line 241 of file TensorStorage.hpp.

```
{
    return m_sizes.size();
}
```

References Nektar::TensorRegions::TensorStorage< DataType >::m_sizes.

6.2.4.4 getState()

```
template<typename DataType >
enum State Nektar::TensorRegions::TensorStorage< DataType >::getState ( )  [inline]
```

Returns whether the storage represents physical solution or spectral/hp element coefficients.

Returns

enum State

Definition at line 294 of file TensorStorage.hpp.

```
{
    return m_state;
}
```

References Nektar::TensorRegions::TensorStorage< DataType >::m_data.

6.2.4.5 operator[]() `[1/2]`

```
template<typename DataType >
DataType& Nektar::TensorRegions::TensorStorage< DataType >::operator[] (
              size_t idx )  [inline]
```

Access an element of the storage container.

Parameters

idx	

Returns

DataType&

Definition at line 281 of file TensorStorage.hpp.

```
        {
            return m_data[idx];
        }
```

References Nektar::TensorRegions::TensorStorage< DataType >::m_data.

6.2.4.6 operator[]() [2/2]

```
template<typename DataType >
DataType Nektar::TensorRegions::TensorStorage< DataType >::operator[] (
            size_t idx ) const [inline]
```

Access an element of the storage container.

This routine provides read-only access to the storage container, which works with const objects.

Parameters

idx	Index to access.

Returns

DataType

Definition at line 294 of file TensorStorage.hpp.

```
        {
            return m_data[idx];
        }
```

6.2.4.7 setState()

```
template<typename DataType >
void Nektar::TensorRegions::TensorStorage< DataType >::setState (
            enum State state = ePhys ) [inline]
```

Sets whether storage is in physical space or coefficient space.

Definition at line 314 of file TensorStorage.hpp.

```
        {
            m_state = state;
        }
```

References Nektar::TensorRegions::TensorStorage< DataType >::m_state.

6.2.4.8 size() [1/2]

```
template<typename DataType >
size_t Nektar::TensorRegions::TensorStorage< DataType >::size ( )  [inline]
```

Returns the total size of the TensorStorage object.

Returns

size_t

Definition at line 252 of file TensorStorage.hpp.

```
        {
            size_t s = 1;
            for (auto &i : m_sizes)
            {
                s *= i;
            }
            return s;
        }
```

References Nektar::TensorRegions::TensorStorage< DataType >::m_sizes.

Referenced by Nektar::TensorRegions::TensorStorage< DataType >::TensorStorage().

6.2.4.9 size() [2/2]

```
template<typename DataType >
size_t Nektar::TensorRegions::TensorStorage< DataType >::size (
              const size_t idx )  [inline]
```

Returns the size of the TensorStorage object.

Parameters

idx	Index of sub-component to request size.

Returns

size_t

Definition at line 268 of file TensorStorage.hpp.

```
        {
            ASSERTL1(idx < m_sizes.size(),
                     "Requested index out of range.");
            return m_sizes[idx];
        }
```

References Nektar::TensorRegions::TensorStorage< DataType >::m_sizes.

6.2.4.10 slice()

```
template<typename DataType >
View Nektar::TensorRegions::TensorStorage< DataType >::slice (
            unsigned int dim,
            const std::vector< size_t > & coord )  [inline]
```

Constructs a TensorStorage::View on a high-dimensional TensorStorage object, corresponding to a single constituent dimension.

Parameters

dim	The dimension to view.
coord	The base coordinate to view.

Returns

View

Definition at line 328 of file TensorStorage.hpp.

```
        {
            ASSERTL0(dim < m_sizes.size(), "dim larger than number of dimensions.");
            ASSERTL0(coord.size() == m_sizes.size(), "coord dimension must match dimension of Tensor.");
            std::shared_ptr<TensorStorage<DataType» ts = this->shared_from_this();
            return std::move(View(ts, dim, coord));
        }
```

References Nektar::TensorRegions::TensorStorage< DataType >::m_sizes.

6.2.5 Member Data Documentation

6.2.5.1 m_data

```
template<typename DataType >
Array<OneD, NekDouble> Nektar::TensorRegions::TensorStorage< DataType >::m_data  [private]
```

Storage for actual data.

Definition at line 337 of file TensorStorage.hpp.

Referenced by Nektar::TensorRegions::TensorStorage< DataType >::getState(), Nektar::TensorRegions::TensorStorage< DataType >::operator[ ](), and Nektar::TensorRegions::TensorStorage< DataType >::TensorStorage().

6.2.5.2 m_sizes

```
template<typename DataType >
std::vector<size_t> Nektar::TensorRegions::TensorStorage< DataType >::m_sizes  [private]
```

Sizes for each sub-component of the tensor.

Definition at line 338 of file TensorStorage.hpp.

Referenced by Nektar::TensorRegions::TensorStorage< DataType >::dims(), Nektar::TensorRegions::TensorStorage< DataType >::size(), and Nektar::TensorRegions::TensorStorage< DataType >::slice().

6.2.5.3 m_state

```
template<typename DataType >
enum TensorStorage::State Nektar::TensorRegions::TensorStorage< DataType >::m_state = ePhys
[private]
```

Flag indicating if data is in coeff or physical state.

Definition at line 338 of file TensorStorage.hpp.

Referenced by Nektar::TensorRegions::TensorStorage< DataType >::setState().

6.3 Nektar::TensorRegions::TensorStorage< DataType >::View Class Reference

Represents a one-dimensional view onto a TensorStorage object.

```
#include <TensorStorage.hpp>
```

Public Member Functions

- View (std::shared_ptr< TensorStorage< DataType >> ts, unsigned int dim, const std::vector< size_t > &coord)

  Construct a new TensorStorage::View object.
- DataType & operator[ ] (size_t idx)

  Access element of a TensorStorage object through the view.
- DataType operator[ ] (size_t idx) const

  Access element of a TensorStorage object through the view.
- void shift (const std::vector< size_t > &s)

  Shifts the view reference coordinate by the given translation vector.
- Array< OneD, DataType > extract ()

  Creates a copy of the data in the viewed dimension.
- void inject (const Array< OneD, DataType > &pData)

  Replaces the data in the TensorStorage object exposed by the view.
- size_t size ()

  Get the size of the view.

Private Member Functions

- void computeOffsetStride ()

  Calculate the member variables m_offset and m_stride from the given reference coordinate m_coord.

Private Attributes

- std::shared_ptr< TensorStorage< DataType > > m_ts

  Underlying TensorStorage object.
- unsigned int m_dim

  The dimension being viewed.
- std::vector< size_t > m_coord

  The reference coordinate for the view.
- size_t m_offset

  Offset of the first entry in the view (based on coordinate).
- size_t m_stride

  Stride for accessing subsequent elements of view.

6.3.1 Detailed Description

template<typename DataType>
class Nektar::TensorRegions::TensorStorage< DataType >::View

Represents a one-dimensional view onto a TensorStorage object.

The object transparently maps onto the original data structure, enabling the underlying TensorStorage solution to be updated through the view. This enables operations in one particular dimension of the tensor. The View object also supports extracting and injecting data into the TensorStorage object through the View.

Template Parameters

DataType	

Definition at line 52 of file TensorStorage.hpp.

6.3.2 Constructor & Destructor Documentation

6.3.2.1 View()

```
template<typename DataType >
Nektar::TensorRegions::TensorStorage< DataType >::View::View (
            std::shared_ptr< TensorStorage< DataType >> ts,
            unsigned int dim,
            const std::vector< size_t > & coord )  [inline]
```

Construct a new TensorStorage::View object.

Parameters

ts	Base TensorStorage object.
dim	The dimension to be viewed.
coord	Coordinate of a point in the required view.

Definition at line 62 of file TensorStorage.hpp.

```
    : m_ts(ts), m_dim(dim), m_coord(coord)
{
    computeOffsetStride();
}
```

References Nektar::TensorRegions::TensorStorage< DataType >::View::computeOffsetStride().

6.3.3 Member Function Documentation

6.3.3.1 computeOffsetStride()

```
template<typename DataType >
void Nektar::TensorRegions::TensorStorage< DataType >::View::computeOffsetStride ( )  [inline],
[private]
```

Calculate the member variables m_offset and m_stride from the given reference coordinate m_coord.

Definition at line 163 of file TensorStorage.hpp.

```
{
    m_stride = 1;
    m_offset = 0;
    for (unsigned int i = 0; i < m_dim; ++i)
    {
        m_offset += m_coord[i] * m_stride;
        m_stride *= m_ts->m_sizes[i];
    }
}
```

References Nektar::TensorRegions::TensorStorage< DataType >::View::m_coord, Nektar::TensorRegions::TensorStorage< DataType >::View::m_dim, Nektar::TensorRegions::TensorStorage< DataType >::View::m_offset, Nektar::TensorRegions::TensorStorage< DataType >::View::m_stride, and Nektar::TensorRegions::TensorStorage< DataType >::View::m_ts.

Referenced by Nektar::TensorRegions::TensorStorage< DataType >::View::shift(), and Nektar::TensorRegions::TensorStorage< DataType >::View::View().

6.3.3.2 extract()

```
template<typename DataType >
Array<OneD, DataType> Nektar::TensorRegions::TensorStorage< DataType >::View::extract ( )
[inline]
```

Creates a copy of the data in the viewed dimension.

This function generates a new Array<OneD, DataType> object and populates it with the view data. No link back to the original TensorStorage is maintained.

Returns

Array<OneD, NekDouble>

Definition at line 118 of file TensorStorage.hpp.

```
{
    Array<OneD, DataType> x(m_ts->size(m_dim));
    for (size_t i = 0; i < m_ts->size(m_dim); ++i) {
        x[i] = m_ts->m_data[m_offset + i*m_stride];
    }
    return x;
}
```

References Nektar::TensorRegions::TensorStorage< DataType >::View::m_dim, Nektar::TensorRegions::TensorStorage< DataType >::View::m_offset, Nektar::TensorRegions::TensorStorage< DataType >::View::m_stride, and Nektar::TensorRegions::TensorStorage< DataType >::View::m_ts.

6.3.3.3 inject()

```
template<typename DataType >
void Nektar::TensorRegions::TensorStorage< DataType >::View::inject (
             const Array< OneD, DataType > & pData )  [inline]
```

Replaces the data in the TensorStorage object exposed by the view.

Parameters

pData	

Definition at line 133 of file TensorStorage.hpp.

```
                {
                    ASSERTL1(pData.size() == m_ts->size(m_dim),
                             "Injected array size does not match dimension size.");
                    for (size_t i = 0; i < m_ts->size(m_dim); ++i) {
                        m_ts->m_data[m_offset + i*m_stride] = pData[i];
                    }
                }
```

References Nektar::TensorRegions::TensorStorage< DataType >::View::m_dim, Nektar::TensorRegions::TensorStorage< DataType >::View::m_offset, Nektar::TensorRegions::TensorStorage< DataType >::View::m_stride, and Nektar::TensorRegions::TensorStorage< DataType >::View::m_ts.

6.3.3.4 operator[]() [1/2]

```
template<typename DataType >
DataType& Nektar::TensorRegions::TensorStorage< DataType >::View::operator[] (
             size_t idx )  [inline]
```

Access element of a TensorStorage object through the view.

This function allows modification of the underlying TensorStorage object.

Parameters

idx	Index in the viewed dimension.

Returns

DataType&

Definition at line 76 of file TensorStorage.hpp.

```
                {
                    return m_ts->m_data[m_offset + idx * m_stride];
                }
```

References Nektar::TensorRegions::TensorStorage< DataType >::View::m_offset, Nektar::TensorRegions::TensorStorage< DataType >::View::m_stride, and Nektar::TensorRegions::TensorStorage< DataType >::View::m_ts.

6.3.3.5 operator[]() [2/2]

```
template<typename DataType >
DataType Nektar::TensorRegions::TensorStorage< DataType >::View::operator[] (
            size_t idx ) const  [inline]
```

Access element of a TensorStorage object through the view.

This function allows read-only access, supporting const TensorStorage objects.

Parameters

idx	

Returns

DataType

Definition at line 89 of file TensorStorage.hpp.

```
                {
                    return m_ts->m_data[m_offset + idx * m_stride];
                }
```

References Nektar::TensorRegions::TensorStorage< DataType >::View::m_offset, Nektar::TensorRegions::TensorStorage< DataType >::View::m_stride, and Nektar::TensorRegions::TensorStorage< DataType >::View::m_ts.

6.3.3.6 shift()

```
template<typename DataType >
void Nektar::TensorRegions::TensorStorage< DataType >::View::shift (
            const std::vector< size_t > & s )  [inline]
```

Shifts the view reference coordinate by the given translation vector.

Parameters

s	Translation vector to shift by.

Definition at line 100 of file TensorStorage.hpp.

```
                {
                    for (unsigned int i = 0; i < m_dim; ++i) {
                        ASSERTL0(m_coord[i] + s[i] < m_ts->size(i),
                                "Requested shift takes coordinate out of range.");
                    }
                    computeOffsetStride();
                }
```

References Nektar::TensorRegions::TensorStorage< DataType >::View::computeOffsetStride(), Nektar::TensorRegions::TensorStorage< DataType >::View::m_coord, Nektar::TensorRegions::TensorStorage< DataType >::View::m_dim, and Nektar::TensorRegions::TensorStorage< DataType >::View::m_ts.

6.3.3.7 size()

```
template<typename DataType >
size_t Nektar::TensorRegions::TensorStorage< DataType >::View::size ( )  [inline]
```

Get the size of the view.

Returns

size_t

Definition at line 147 of file TensorStorage.hpp.

```
{
    return m_ts->size(m_dim);
}
```

References Nektar::TensorRegions::TensorStorage< DataType >::View::m_dim, and Nektar::TensorRegions::TensorStorage< DataType >::View::m_ts.

6.3.4 Member Data Documentation

6.3.4.1 m_coord

```
template<typename DataType >
std::vector<size_t> Nektar::TensorRegions::TensorStorage< DataType >::View::m_coord  [private]
```

The reference coordinate for the view.

Definition at line 155 of file TensorStorage.hpp.

Referenced by Nektar::TensorRegions::TensorStorage< DataType >::View::computeOffsetStride(), and Nektar::TensorRegions::TensorStorage< DataType >::View::shift().

6.3.4.2 m_dim

```
template<typename DataType >
unsigned int Nektar::TensorRegions::TensorStorage< DataType >::View::m_dim  [private]
```

The dimension being viewed.

Definition at line 154 of file TensorStorage.hpp.

Referenced by Nektar::TensorRegions::TensorStorage< DataType >::View::computeOffsetStride(), Nektar::TensorRegions::TensorStorage< DataType >::View::extract(), Nektar::TensorRegions::TensorStorage< DataType >::View::inject(), Nektar::TensorRegions::TensorStorage< DataType >::View::shift(), and Nektar::TensorRegions::TensorStorage< DataType >::View::size().

6.3.4.3 m_offset

```
template<typename DataType >
size_t Nektar::TensorRegions::TensorStorage< DataType >::View::m_offset  [private]
```

Offset of the first entry in the view (based on coordinate).

Definition at line 156 of file TensorStorage.hpp.

Referenced by Nektar::TensorRegions::TensorStorage< DataType >::View::computeOffsetStride(), Nektar::↩ TensorRegions::TensorStorage< DataType >::View::extract(), Nektar::TensorRegions::TensorStorage< DataType >::View::inject(), and Nektar::TensorRegions::TensorStorage< DataType >::View::operator[ ]().

6.3.4.4 m_stride

```
template<typename DataType >
size_t Nektar::TensorRegions::TensorStorage< DataType >::View::m_stride  [private]
```

Stride for accessing subsequent elements of view.

Definition at line 157 of file TensorStorage.hpp.

Referenced by Nektar::TensorRegions::TensorStorage< DataType >::View::computeOffsetStride(), Nektar::↩ TensorRegions::TensorStorage< DataType >::View::extract(), Nektar::TensorRegions::TensorStorage< DataType >::View::inject(), and Nektar::TensorRegions::TensorStorage< DataType >::View::operator[ ]().

6.3.4.5 m_ts

```
template<typename DataType >
std::shared_ptr<TensorStorage<DataType> > Nektar::TensorRegions::TensorStorage< DataType >↩
::View::m_ts  [private]
```

Underlying TensorStorage object.

Definition at line 153 of file TensorStorage.hpp.

Referenced by Nektar::TensorRegions::TensorStorage< DataType >::View::computeOffsetStride(), Nektar::↩ TensorRegions::TensorStorage< DataType >::View::extract(), Nektar::TensorRegions::TensorStorage< Data↩ Type >::View::inject(), Nektar::TensorRegions::TensorStorage< DataType >::View::operator[ ](), Nektar::Tensor↩ Regions::TensorStorage< DataType >::View::shift(), and Nektar::TensorRegions::TensorStorage< DataType >↩ ::View::size().

Chapter 7

File Documentation

7.1 TensorRegion.cpp File Reference

```
#include <MultiRegions/ExpList.h>
#include <TensorRegions/TensorRegion.h>
```

Namespaces

- Nektar
- Nektar::TensorRegions

7.2 TensorRegion.h File Reference

```
#include <LibUtilities/BasicUtils/SharedArray.hpp>
#include <MultiRegions/ExpList.h>
#include <TensorRegions/TensorStorage.hpp>
```

Classes

- class Nektar::TensorRegions::TensorRegion

  High-dimensional region, comprising of a tensor-product of two MultiRegions.

Namespaces

- Nektar
- Nektar::TensorRegions

Typedefs

- using Nektar::TensorRegions::TensorRegionStorage = TensorStorage< NekDouble >

  Double-precision floating-point specialisation of a TensorStorage.
- using Nektar::TensorRegions::TensorRegionStorageSharedPtr = std::shared_ptr< TensorRegionStorage >

  Shared pointer to a double-precision floating-point TensorStorage object.
- using Nektar::TensorRegions::TensorRegionSharedPtr = std::shared_ptr< TensorRegion >

  A shared pointer to a TensorRegion object.

7.3 TensorRegions.hpp File Reference

```
#include <vector>
```

Namespaces

- Nektar
- Nektar::TensorRegions

7.4 TensorRegionsDeclspec.h File Reference

Macros

- #define TENSOR_REGIONS_EXPORT

7.4.1 Macro Definition Documentation

7.4.1.1 TENSOR_REGIONS_EXPORT

```
#define TENSOR_REGIONS_EXPORT
```

Definition at line 42 of file TensorRegionsDeclspec.h.

7.5 TensorStorage.hpp File Reference

```
#include <memory>
#include <LibUtilities/BasicUtils/SharedArray.hpp>
```

Classes

- class Nektar::TensorRegions::TensorStorage< DataType >

  Storage container for TensorRegions solution.
- class Nektar::TensorRegions::TensorStorage< DataType >::View

  Represents a one-dimensional view onto a TensorStorage object.

Namespaces

- Nektar
- Nektar::TensorRegions
- TensorRegions

  Namespace encapsulating algorithms and data structures for constructing high-dimensional high-order finite element spaces.

Typedefs

- template<typename DataType >
  using Nektar::TensorRegions::TensorStorageSharedPtr = std::shared_ptr< TensorStorage< DataType > >

  A shared pointer to a TensorStorage object.