Q2 - Tensors

Background

Tensors

Note: a lot of this question can be done purely in terms of standard knowledge of matrices.

A Tensor is a generalisation of the ideas of scalars, vectors and matrices to larger dimensions. Tensors are widely found in physics and electrodynamics, and more recently have been used as the underlying primitive for machine learning frameworks such as Tensorflow.

Scalars, column vectors, and matrices are special-cases of tensors:

- Scalar : A single number.
- Vector : 1D array of numbers with a single index.
- Matrix : A 2D matrix (array) of numbers accessed using two indices.
- Tensor : An order-n tensor is an n-dimensional array of numbers accessed using n indices.

Each order-n tensor has `n` dimension lengths, with each length describing the size of the tensor in that dimension. The length of a dimension may have length 1 (unitary), but zero length dimensions are not allowed.

For simplicity we will **not** consider tensors of order less than 2, and scalars, row vectors, and column vectors will be represented as order-2 matrices with one or two unitary dimensions.

- Scalar (order-2) : Dimensions are `[1,1]`
- ColVec (order-2) : Dimensions are `[n,1]`, where `n>=1` gives the number of rows.
- RowVec (order-2) : Dimensions are `[1,m]`, where `m>=1` gives the number of columns
- Matrix (order-2) : Has two dimensions, describing the number of columns and rows. Its dimensions are `[n,m]`, where `n>=1` and `m>=1`
- Tensor (order-n) : Has n dimensions, with a length specified for each of the n dimensions. Its dimensions are `[d_1,d_2,...,d_n ]`, where `d_i>=1`

Given an order-n tensor `D` with dimensions `[d_1,d_2,...,d_n]` we will use the notation `D[i_1,i_2,...,i_n]` to represent the value at index `i_1,i_2,...,i_n`, where `0 <= i_j < d_j` for `1 <= j <= n`.

Tensor operations

Tensor addition

Tensors addition is a generalisation of matrix addition:

- Scalar + Scalar : `C[0,0] = A[0,0] + B[0,0]`
- ColVec + ColVec : `C[i,0] = A[i,0] + B[i,0]`
- RowVec + RowVec : `C[0,i] = A[0,i] + B[0,i]`
- Matrix + Matrix : `C[i,j] = A[i,j] + B[i,j]`
- Tensor3 + Tensor3 : `C[i,j,k] = A[i,j,k] + B[i,j,k]`
- TensorN + TensorN : `C[d_1,...,d_n] = A[d_1,...,a_n] + B[d_1,...,d_n]`

The indices `i,j,k`, or `d_1,...,d_n` range over the length of each object in that dimension.

Mathematically we require that the objects being added have the same dimensions. However, there is a common convention in machine-learning that says that unitary (length-1) dimensions can be "broadcast" to match non-unitary dimension, so that they produce a consistent result. This allows for common operations such as adding a scalar to a matrix:

- Scalar + ColVec : `C[i,0] = A[0,0] + B[i,0]`
- ColVec + RowVec : `C[i,j] = A[i,0] + B[0,j]`
- Matrix + Scalar : `C[i,j] = A[i,j] + B[0,0]`
- Matrix + ColVec : `C[i,j] = A[i,j] + B[i,0]`
- Matrix + RowVec : `C[i,j] = A[i,j] + B[0,j]`
- Tensor3 + Scalar : `C[i,j,k] = A[i,j,k] + B[0,0]`
- Tensor3 + Matrix : `C[i,j,k] = A[i,j,k] + B[i,j]`
- Tensor4 + Tensor3 : `C[i,j,k,m] = A[i,j,k,m] + B[i,j,k]`

Tensor multiplication

For our "normal" tensors we have the following multiplications between different dimension objects:

- Scalar * Scalar : `C[0,0]  = A[0,0] * B[0,0]`
- ColVec * Scalar : `C[i,0]  = A[i,0] * B[0,0]`
- RowVec * ColVec : `C[0,0]  = sum( A[0,k] * B[k,0], k)`
- ColVec * RowVec : `C[i,j]  = A[i,0] * B[0,j]`
- Matrix * ColVec : `C[i,0]  = sum( A[i,k] * B[k,0], k)`
- Matrix * Matrix : `C[i,j]  = sum( A[i,k] * B[k,j], k)`

The indices i,j, and k range over the length of each object in that dimension. Where the index k appears in both A and B, the lengths must match, as expected for normal vector and matrix operations. Note that the cases where there is no explicit `sum` are just summations over a single element, and so are still degenerate sums.

Note: full implementation of tensor multiplication is complicated, but is only relevant for half of one subquestion. It is worth proceeding based on normal matrices first.

There are a number of possible ways of defining general tensor multiplication, with different mathematical and coding frameworks choosing different conventins.

Here we use one of the simplest definitions: given two tensors `A` (of order `a_1..a_n`) and `B` (of order `b_1..b_m`), we define multiplication `C = A * B` as follows:

```
C[a_1, ...., a_{n-1} , b_2, ..., b_m] = sum( A[a_1, ...a_{n-1}, k] * B[ k, b_2, ..., b_m], k )
```

The `k` dimension length in each tensor (last dimension in A, first dimension in B) must match, as with standard matrix/vector multiplication. This definition of multiplication matches all of our standard notions of matrix/vector/scalar multiplication.

Applying these rules, we can define an order-3 tensor times a matrix:

- Tensor3 * Matrix : `C[a_1,a_2, b_2] = sum( A[a_1,a_2, k] * B[k, b_2], k )`

As pseudo-code, this `Tensor3 * Tensor2` multiplication might look like:

```
for ia1 = 0..a_1-1
  for ia2 = 0..a_2-1
    for ib2 = 0..b_2-1
      C[ia0,ia1,ib2] = 0
      for k = 0..a_3-1
        C[ia0,ia1,ib2] = C[ia0,ia1,ib2] + A[ia0,ia1,k] * A[k,ib2]
```

Or an order-3 tensor times an order-3 tensor:

- Tensor3 * Tensor3 : `C[a_1,a_2, b_2,b_3] = sum( A[a_1,a_2, k] * B[k, b_2,b_3], k)`

Tensor Representation in C++

The class `Tensor` in `tensor.hpp` represents an abstract tensor of any order and dimensions. The member `size()` returns a vector describing the length in each dimension. The number of elements in the `size()` vector gives the order of the tensor, regardless of how many unitary (length-1) dimensions there are. So for example, a "scalar" tensor would return dimensions `[1,1]`.

Indices

Note: the behaviour of indices is flexible, which makes them complex. You may wish to implement the "simple" behaviour first, then come back to complex indices as time allows.

As a convenience measure, when reading or writing the vector it is possible to give an index which is too small or too large, as long as certain constraints are met:

- The index may be shorter (contain fewer values) than the tensor's dimensions. In this case, the index is implicitly extended with 0 until it matches the tensor dimension.
- The index may be longer (contain more values) than the tensor's dimensions. In this case the extra dimensions must all specify offset 0, otherwise the index is not valid.

In addition, when **reading** from a tensor we allow for broadcasting of the dimension:

- If the length of a dimension is 1, then any offset in that dimension will be "broadcast", and map to offset 0 in that dimension. For broadcast purposes, all tensors are assumed to be unitary in all dimensions higher than their order.

For example, if we have a dimension `[4,5]` matrix we could read or write it at the index `[2,3]`, so the index order matches the matrix order. However, we could also read or write it at `[1]`, which would be implicitly extended to `[1,0]`, or we could index it using `[3,2,0,0]`, which would be implicitly contracted to `[3,2]`. If we were trying to read the tensor at index `[2,3,2]`, then the extra index `2` would be implicitly broadcast, resulting in `[2,3]`; indices `[2,3,4]` and `[2,3,100]` would also result in the same element at `[2,3]`.

Any out of range indices are invalid, such as `[6,7]` which exceeds the size of the tensor, or writing to `[2,3,2]` as the extra dimension's offset is non-zero. Implementations may respond in any way to invalid accesses, including crashing. It is often helpful to use assertions to detect the precise point at which an invalid index is used, though this is not required.

The following table gives examples of this matrix dimensions, indices, and how they map for read and write. Note that this was compiled manually, and while it has been checked multiple times, there is an outside chance of error.

Matrix	Index	Maps to (read)	Maps to (write)
`[1,1]`	`[0,0]`	`[0,0]`	`[0,0]`
`[1,1]`	`[0]`	`[0,0]`	`[0,0]`
`[1,1]`	`[]`	`[0,0]`	`[0,0]`
`[1,1]`	`[0,0,0]`	`[0,0]`	`[0,0]`
`[1,1]`	`[0,3]`	`[0,0]`	Invalid
`[1,1]`	`[2,0]`	`[0,0]`	Invalid
`[3,1]`	`[2,0]`	`[2,0]`	`[2,0]`
`[3,1]`	`[2,0,0]`	`[2,0]`	`[2,0]`
`[3,1]`	`[2,1]`	`[2,0]`	Invalid
`[3,1]`	`[4,1]`	Invalid	Invalid
`[3,1]`	`[2,0,0,0]`	`[2,0]`	`[2,0]`
`[3,1]`	`[2,0,0,1]`	`[2,0]`	Invalid
`[4,4]`	`[3,2]`	`[3,2]`	`[3,2]`
`[4,4]`	`[3,2,0]`	`[3,2]`	`[3,2]`

Matrix	Index	Maps to (read)	Maps to (write)
`[4,4]`	`[3]`	`[3,0]`	`[3,0]`
`[4,4]`	`[3,3,0,0]`	`[3,3]`	`[3,3]`
`[4,4]`	`[3,3,0,1]`	`[3,3]`	Invalid

The compiler-defined type `size_t` is used for indices and lengths. The type `int` may only be 32-bits wide for efficiency reasons, but in a 64-bit machine we can allocate arrays much larger than $2^{31} - 1$ elements. `size_t` is built-in unsigned integer type which is guaranteed to be able to hold indexes into the longest array that can be allocated.

Testing tensor maths

The program `test_tensor_maths.cpp` provides minimal test-cases for addition and subtraction. These test-cases are far from complete, and each case took a few minutes to generate manually. However, manual calculation greatly increased understanding of how multiplication works, plus considering edge cases found a number of errors.

You are welcome to modify/extend/copy this program, **or** you might find it easier to start from scratch if you find it difficult to understand.

Deliverables

In this question you should assume that any programs will be compiled using exactly three source files: `tensor_core.cpp`, `tensor_maths.cpp`, and the program's source file containing main.

T1 : Tensor operations (25%)

T1.1 : Add a function `Tensor::order` (5%)

Add a new non-`virtual` function called `order` to the `Tensor` base-class which returns the order of the tensor as an integer.

The definition can be added in either the `tensor.hpp` header or in `tensor.cpp`.

T1.2 : Implement `is_vector` (5%)

Implement the function `is_vector`. This returns `true` if a given tensor has at most one non-unitary dimension.

Node: a tensor of any order might be a vector; it depends on it's dimensions.

T1.3 : Implement `Tensor::volume()` (5%)

Modify the member function `volume` in `tensor.cpp` so that it returns the total number of numeric elements in a tensor, based on the current `size`.

T1.4 : Implement `offset_to_index` (10%)

Implement the function `offset_to_index`. This is defined to be the inverse of `index_to_offset(index,false)`.

The existing function `index_to_offset` takes the dimensions of a tensor plus an element index within those dimensions, and returns a unique non-zero linear offset for that element.

T2 : Higher-order tensors (40%)

T2.1 : Create an order-3 tensor (10%)

Create a class called `Order3Tensor` in a header `order_3_tensor.hpp` which can represent order-3 tensors. Its constructor should take one parameter of type `vector<size_t>` giving the dimensions of the vector, which will always be of length 3.

You can use any parts of `MatrixTensor` as a starting point.

T2.2 : Create a generalised tensor (15%)

Create a class called `OrderNTensor` in a header `order_n_tensor.hpp` which can represent order-n tensors, so it can handle tensors with any number of dimensions.

Marks are available at two levels of functionality:

1. (7.5%) Tensors which are able to read and write values at any index, but cannot be resized once created.
2. (7.5%) Tensors which also correctly implement `Tensor::resize`.

You can use any parts of `MatrixTensor` as a starting point.

T2.3 : Complete the factory function `create_tensor` (5%)

Complete the factory function `create_tensor` so that it is able to create tensors of any dimensions. It should automatically choose the most specialised implementation (matrix, order-3, or order-n) based on the dimensions it is given.

Note: this is assessed independently of whether the returned class is fully working or not, as long as it returns a class of the right type.

T2.4 : Create a test for order-n tensors (10%)

Create a test-bench `test_create_tensors.cpp` which tests tensor implementations from order 2 to order 5.

For each tensor order `i` from 2 to 5 (inclusive), the test-bench should:

1. Instantiate a tensor of order `n` with length 2 in each of the first `n` dimensions using `create_tensor`.
2. Set every element of the tensor to a unique non-zero value using `Tensor::write`. No element in the tensor should receive the same value.
3. Check every element of the tensor has the same unique non-zero value using `Tensor::read`.
4. Print `Pass i` followed by a new-line to stdout, where i is the order of the tensor tested.

If all orders 2 through 5 succeed, then the program should print `Success` and return the success code.

For example, if only the matrix implementation works, then the output would be:

```
$ ./build_test_create_tensors.sh
$ ./test_create_tensors
Pass 2
$
```

While if all the implementations work the output would be:

```
$ ./test_create_tensors
Pass 2
Pass 3
Pass 4
Pass 5
Success
```

Hint: you may find ***`first_index`*** *and* ***`next_index`*** *useful. Hint: you may find* ***`index_to_offset`*** *useful.*

You do not need to test the functionality of resize.

Your test should not rely on any implementation details of the classes, and should work even if it is compiled against a different tensor implementation.

The test-bench can abort or crash in any way if the tensor implementation does not work, but should only print `Pass i` if the implementation passed for the given order `i`.

T3 : Tensor arithmetic (35%)

T3.1 : Tensor addition (15%)

Complete the implementation of `add`. Two levels of functionality are recognised:

1. (7.5%) Non-broadcast: the ability to add tensors of the same dimensions.
2. (7.5%) Broadcast: the ability to add tensors while broadcasting any dimensions as necessary.

Hint: given a set of dimensions, the functions ***`first_index`****+****`next_index`*** *can be used to iterate over every index within those dimensions.*

T3.2 : Tensor multiplication (20%)

Complete the implementation of `multiply`.

You are only required to support the case where the left matrix has greater or equal order than the right matrix.

Two levels of functionality are recognised:

1. (10%) Order-2 : Order-2 tensors (matrices) can be multiplied with order-2
2. (10%) Order-3 : Order-3 tensors can be multiplied with order-3 and order-2

You are not required to support left-hand side tensors of order less than 2 or greater than 3.

Hint: try to implement lower-orders before higher-orders.

Hint: you may wish to write multiple functions, and dispatch to them based on the order.