Cog Tensor Field

External Reference Specification

Version 1

Greg Snider

Summary: Definition of tensor fields, how they are described, and how they are laid out in memory. This is needed by writers of Sensors and Actuators.

# 1. Overview

A tensor field is the data abstraction of Cog. It is defined to be a multidimensional array (the field) of multidimensional arrays (the tensors) of numbers. All tensors in a given field have exactly the same size and shape.

The field itself has a *dimensionality* equal to the number of dimensions of the array containing it. The number of dimensions is limited to 0, 1, 2, or 3. A zero-dimensional field holds a single tensor.

A tensor in a field has an *order* equal to the number of dimensions of the array containing it. The order is currently limited to 0, 1, or 2. An order-0 tensor (which holds a single number) is called a *scalar*, an order-1 tensor is called a *vector*, and an order-2 tensor is called a *matrix*. The numbers in a tensor are 32 bit floating point numbers (may be extended).

# 2. Tensor Field Descriptor

Since a tensor field must be accessible to both the Cog Native Runtime (which has a C interface) and the Cog Compiler (written in Scala), there are parallel structures for describing them.

*C version:*

struct CogTensorField {

int index; // Index assigned by Cog compiler.

char\* userName; // User-defined name for the field, using probe().

int fieldDimensions; // Number of field dimensions: 0, 1, 2 or 3.

int fieldShape[]; // Size of each field dimension.

int tensorOrder; // Order (dimensionality) of each tensor: 0, 1 or 2.

int tensorShape[]; // Size of each tensor dimension.

enum CogNumberType numberType; // Type of numbers held in each tensor.

enum CogTensorFieldType fieldType; // Type of tensor field.

}

enum CogNumberType {

COG\_FLOAT\_16, // Half float (not supported in Scala or C)

COG\_FLOAT\_32, // Float

COG\_FLOAT\_64, // Double

COG\_INT\_8, // 8 bit signed integer

COG\_INT\_16, // 16 bit signed integer

COG\_INT\_32, // 32 bit signed integer

COG\_INT\_64 // 64 bit signed integer

}

enum CogTensorFieldType {

COG\_SENSOR\_FIELD,

COG\_ACTUATOR\_FIELD,

COG\_STATE\_FIELD,

COG\_HIDDEN\_FIELD

}

*Scala version:*

class CogTensorField {

val index: Int

val username: String

def fieldDimensions = fieldShape.length

val fieldShape: Array[Int]

def tensorOrder = tensorShape.length

val tensorShape: Array[Int]

val numberType: CogNumberType

val fieldType: CogFieldType

}

sealed abstract class CogNumberType

CogFloat16 extends CogNumberType

CogFloat32 extends CogNumberType

CogFloat64 extends CogNumberType

CogInt8 extends CogNumberType

CogInt16 extends CogNumberType

CogInt32 extends CogNumberType

CogInt64 extends CogNumberType

sealed abstract class CogTensorFieldType

CogSensorField extends CogTensorFieldType

CogActuatorField extends CogTensorFieldType

CogStateField extends CogTensorFieldType

CogHiddenField extends CogTensorFieldType

# 3. Dimension Naming

Cog uses a (layer, row, column) coordinate system rather than an (x, y, z) coordinate system for fields and tensors. A 0-dimensional field has no coordinates at all. A 1-dimensional field uses a (column) coordinate and is of size “columns.” A 2-dimensional field uses (row, column) coordinates and is of size “rows” x “columns.” A 3-dimensional field uses (layer, row, column) coordinates and is of size “layers” x “rows” x “columns.”

Tensors use the same (layer, row, column) convention for representing their size and shape, and for indexing numbers within them.

# 4. Memory Layout

Tensor fields are flattened in buffers that are exchanged between a Cog application and the Runtime. The layout generally follows the C convention of row-major order for scalar fields, but with some twists for non-scalar fields.

#### Scalar field layout

Straightforward, row-major layout. The final dimension varies the fastest as a linear buffer is traversed, while the first dimension varies most slowly.

#### Vector field layout

The vector field is conceptually “sliced” by vector index to create scalar field “planes.” These planes are concatenated in memory using the scalar field layout. For example, a vector field with length 3 vectors is sliced into three scalar fields, the first corresponding to element 0 of every vector in the field, the second corresponding to element 1 of every vector in the field, and the third corresponding to element 2 of every vector in the field.

#### Matrix field layout

Matrix fields are also sliced into scalar field planes and concatenated in buffer memory. The indices of the matrix are traversed in row-major order to create the planes. For example, a matrix field holding 2 x 2 matrices would have four scalar field slices: the first corresponding to matrix elements (0, 0), the second to matrix elements (0, 1), the third to matrix elements (1, 0), and the fourth to matrix elements (1, 1).