LINEAR ALGEBRA NOTES FOR INTRO. SCIENTIFIC COMPUTING

DAVID M. ROGERS

1. Definitions

1.1. Tensors. A tensor is just a block of numbers. Like anything computational, it's meaning depends on how you use it. These notes describe the peculiar interpretation called linear algebra - where subsets of n consecutive numbers in a tensor are viewed as points in n-dimensional space. The space could correspond to the familiar 3D world (each atom is a point in $\mathbb{R}^3$ space) or to abstract spaces, like the O(1 million) pixels in an image (each image is a point in $\mathbb{R}^{10^6}$ space) or to a set of 12 "basis" functions (each function in the space is a point in $\mathbb{R}^{12}$).

Tensor spaces are defined by their shape. The order of a tensor is the number of indices. Low-order tensors have special names:

- A scalar is just a number. It has no indices, and so is an order-zero tensor – in the space T_0.
- A vector with m real components is in the space $\mathbb{R}^m$. These vectors are the building blocks of linear algebra. They represent points in m-dimensional space, and higher-order tensors are thought of as blocks of vectors.
  An empty vector can be declared in python using, when $m = 12$,
  `from numpy import *; x = zeros(12)`. These are also known as null-vectors, since they remain zero with linear transformations, and because null (ϕ) is a mathematical term denoting the empty set.
- A vector with m complex components is in $\mathbb{C}^m$ (since $\mathbb{C}$ conventionally stands for the set of complex numbers)
  Both of these are in an order-1 tensor space, T_m.
- An $n \times m$ matrix of real numbers (pronounced n rows by m columns) is in the space $\mathcal{M}_{n,m}$, and can equally be said to be $T_{n,m}$.

More general tensor spaces can be referred to using a list of numbers, indicating the shape. Most of the numpy routines that build vectors allow you to build tensors of an arbitrary shape as well. For example 2 sets of 4 points in 3D space could be represented as a tensor in the space $T_{2,4,3}$. An empty (zero) tensor with this shape can be declared using

```
>>> M = zeros((2,3,4), dtype=float64)
```

Each index moves along a different dimension. In python, the dimensions are numbered consecutively from 0 – so the last index is dimension 2. The last index holds the individual vectors. The ordering of the indices depends entirely on how you have chosen to address

your tensor. For example, if groups are along dimension 0, and sub-groups are along dimension 1, you can address the second vector from the first group as `M[0,1]`

1.2. **Inner Product.** The inner product (also known as the dot product) on 2 vectors is defined as

$$x \cdot y = \sum_{i=0}^{m-1} x_i y_i \tag{1}$$

It takes two vectors in the same space – here $x \in \mathbb{R}^m, y \in \mathbb{R}^m$, and returns a number.

In python (using numpy), the dot product can be written,

```
>>> x = arange(12); y = ones(12) # test data
>>> dot(x,y) # evaluates to 12(12-1)/2
>>> sum(x*y) # equivalent
```

1.3. **Vector Relations.** The norm of a vector is just

$$|x| \equiv \sqrt{x \cdot x} \tag{2}$$

A vector with a unit norm is said to be normalized. You can normalize any vector by dividing by its magnitude, that is $x \to x/|x|$.

The inner product is the basis for defining the shortest angle between two vectors, using

$$\cos\theta_{xy} = \frac{x \cdot y}{|x||y|} \tag{3}$$

This is a general definition for multi-dimensional vectors.

Two vectors that meet at right-angles are said to be perpendicular, or orthogonal. Mathematically, this means

$$0 = |x||y|\cos\theta_{xy} = x \cdot y. \tag{4}$$

Viewed algebraically for two vectors $x, y \in R^n$, this is a linear equation in $2n$ variables. It can be used to determine y_0 given x and all $y_1, \cdots y_{n-1}$. You can also make a vector y orthogonal to x by subtracting its projection along x, which is $y \to y - x x \cdot y/|x|^2$.

For complex vectors ($\in \mathbb{C}^n$), these definitions break down a little bit. The norm can be salvaged by re-defining the inner product as $x \cdot y = \sum_i x_i^* y_i$, which at least gives every vector a positive norm. This is not the same as numpy's `dot`.

1.4. **Matrix Multiplication.** The matrix-vector product between $A \in \mathcal{M}_{n,m}$ and a vector, $x \in \mathbb{R}^m$ is:

$$b_i = (A \cdot x)_i = \sum_{j=0}^{m-1} A_{ij} x_j, \quad i = 0, 1, \ldots, n-1 \tag{5}$$

This can be visualized by breaking up the $n \times m$ matrix, A, into rows as

$$\begin{bmatrix} b[0] \\ b[1] \\ \vdots \\ b[n-1] \end{bmatrix} = \begin{bmatrix} A[0,:] \\ \hline A[1,:] \\ \hline \vdots \\ A[n-1,:] \end{bmatrix} \cdot x = \begin{bmatrix} A[0,:] \cdot x \\ A[1,:] \cdot x \\ \vdots \\ A[n-1,:] \cdot x \end{bmatrix} \tag{6}$$

This comes from thinking about turning the matrix, A, and attaching it to x.

You can equivalently think about turning the vector, x, and attaching it to the top dimension of A. Each element of the vector hits a different column of the matrix, so

$$\begin{aligned} b &= \left[A[:,0] \,\middle|\, A[:,1] \,\middle|\, \cdots \middle| A[:,m-1]\right] \cdot \begin{bmatrix} x[0] \\ x[1] \\ \vdots \\ x[m-1] \end{bmatrix} \\ &= A[:,0]x[0] + A[:,1]x[1] + \cdots + A[:,m-1]x[m-1] \end{aligned} \tag{7}$$

The matrix-matrix product is defined by:

$$Y_{ij} = (A \cdot X)_{ij} = \sum_k A_{ik} X_{kj} = A[i,:] \cdot X[:,j] \tag{8}$$

Each element of Y comes from a row of A and a column of X. Obviously, the last dimension of A has to match the first dimension of X. If $A \in \mathcal{M}_{a,b}$ and $X \in \mathcal{M}_{b,c}$, then $Y \in \mathcal{M}_{a,c}$ (the output matrix has the first and last dimensions of matrices A and X, respectively). You can visualize this by grouping the left matrix into rows and the right into columns,

$$Y = \begin{bmatrix} A[0,:] \\ \hline A[1,:] \\ \hline \vdots \\ \hline A[a-1,:] \end{bmatrix} \left[X[:,0] \,\middle|\, X[:,1] \,\middle|\, \cdots \,\middle|\, X[:,c-1]\right]. \tag{9}$$

2. Linear Transformations

Linear transformations express a scale and a rotation of a vector about the origin. Because we can build objects out of vectors, we only have to consider how linear transformations act on one vector at a time. The linear transformation of a vector, v, is

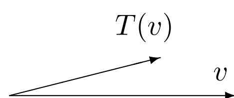
$$T_A(v) = \sum_j A_{ij} v_j \tag{10}$$

where the coefficients, A_{ij} are arbitrary, but can't depend on v. The linear transformation is defined to be linear in v – so that (for a scalar [i.e. a number] α)

$$T_A(\alpha v) = \alpha T_A(v) \tag{11}$$

and

$$T_A(v + w) = T_A(v) + T_A(w). \tag{12}$$



Notice that the linearity properties (Eq. 11 and Eq. 12) also apply to both sides of the dot product (Eq. 1). Because the dot product was used to define matrix-vector and matrix-matrix multiplication, both of those products are also linear in the left and right sides.

Translations are not linear transformations, since they do not have the properties above. However, they can be expressed as linear transformations if we bend our minds a little to re-define points. This is not covered here, but will be the subject of the lecture on homogeneous coordinates.

How do we figure out the coefficient matrix, A_{ij} for a linear transformation? We reason as follows. The coefficients will tell us how to transform a general vector,

$$v = v_0 \hat{x}_0 + v_1 \hat{x}_1 + \cdots + v_{m-1} \hat{x}_{m-1}. \tag{13}$$

So we need only ask *what the linear transformation does to each of the* $\hat{x}_i$*-s*, since

$$L_A(v) = v_0 L_A(\hat{x}_0) + v_1 L_A(\hat{x}_1) + \cdots + v_{m-1} L_A(\hat{x}_{m-1}) \tag{14}$$

The $\hat{x}_i$-s can be any set of directions that allow us to write down a general vector using Eq. 13. A particularly simple choice is given by the *global frame*,

$$\left[\hat{x}_0 \,\middle|\, \hat{x}_1 \,\middle|\, \cdots \,\middle|\, \hat{x}_{n-1} \right] = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} \equiv I \tag{15}$$

The last $\equiv$ sign notes that the identity matrix, I (`identity(N)` in numpy) is defined by a matrix with ones along the diagonal.

The similarity between Eqns. 13 and 7 is the reason that sets of "basis vectors" are often written as

$$A = \left[\hat{x}_0 \,\middle|\, \hat{x}_1 \,\middle|\, \cdots \,\middle|\, \hat{x}_{n-1} \right]. \tag{16}$$

The matrix A describes a coordinate frame.

3. Inverses

If we have a linear transformation from v to $T_A(v)$, we might ask about the reverse of this operation. How do we get from a transformed vector back to the original? In general, it isn't possible to go back - since A may have been a matrix of zeros, or something awful

like that. However, if it is possible to go back, then the linear transformation from x to $y = Ax$ can be un-done by inverting the matrix, $x = A^{-1}y$. The existence and uniqueness requirements for this inverse transformation are essentially[1] the same as the existence and uniqueness of the matrix inverse.

For changes in coordinate frames, (Eq. 16), the matrix inverse is very easy to find, since

$$A^T \cdot A = \begin{bmatrix} \hat{x}_0^T \\ \hline \hat{x}_1^T \\ \hline \vdots \\ \hline \hat{x}_{n-1}^T \end{bmatrix} \cdot \left[\hat{x}_0 \,\middle|\, \hat{x}_1 \,\middle|\, \cdots \,\middle|\, \hat{x}_{n-1}\right] = I = A^{-1}A \tag{17}$$

so

$$A^{-1} = A^T \tag{18}$$

This only happens when $\hat{x}_i \cdot \hat{x}_j = 0$ for $i \neq j$ and $\hat{x}_i \cdot \hat{x}_j = 1$ - a so-called orthonormal set of basis vectors.

4. Generalization to Tensors

We have seen that the matrix-matrix product is defined using dot products between vectors composing the two matrices. What if we change the addressing of the matrices, or want to do multiple matrix-matrix products between sets of matrices? The general case that handles all of these is called the tensor contraction.

When two tensors are multiplied together, say $A \in T_{n_0,n_1,\ldots,n_{k-1}}$, and $B \in T_{m_0,m_1,\ldots,m_{p-1}}$, we now have lots of possible products over $k + p$ indices,

$$C_{i_0,i_1,\ldots,i_{k+p-1}} = A_{i_0,i_1,\ldots,i_{k-1}} B_{i_k,i_{k+1},\ldots,i_{k+p-1}} \tag{19}$$

This is the complete tensor outer product. No contraction is performed, and
$C \in T_{n_0,\ldots,n_{k-1},m_0,\ldots,m_{p-1}}$.

For two vectors, $A \in T_n$ and $B \in T_m$, this outer product generates a matrix $C \in T_{n,m}$. The *other* way to combine vectors is the inner product. The inner product pairs the indices of A and B and sums, so n has to equal m.

A general tensor contraction pairs some of the indices, and takes an outer product over the rest of the indices. Labeling the indices that are summed with lowercase letters, and the indices that are not summed with uppercase letters makes it simple to express tensor contractions. Here are some examples, along with the corresponding numpy code using tensors filled with ones. Of course you have to declare Ni, etc. as the size of each array.

- A scalar-vector product (inner or outer) $c_I = ab_I$
  ```
  >>> c = 1*ones(Ni)
  ```
- A vector-vector inner product, $c = a_i b_i$
  ```
  >>> c = tensordot(ones(Ni), ones(Ni), [0,0])
  ```
  (or any of the methods above for the dot product)

---

[1]I say essentially, since the actual best way to go back is the matrix pseudoinverse, $A^\dagger =$ `numpy.linalg.pinv(A)`, but the pseudoinverse is essentially the inverse anyway.

- A matrix-vector product, $c_I = A_{Ij} b_j$

```
>>> c = tensordot(ones((Ni,Nj)), ones(Nj), [1,0])
```

- A matrix-matrix product: $C_{IJ} = A_{Ik} B_{kJ}$ (for $C = A \cdot B$)

```
>>> C = tensordot(ones((Ni,Nk)), ones((Nk,Nj)), [1,0])
```

- A matrix-matrix product including a transpose: $C_{IJ} = A_{kI} B_{kJ}$ (for $C = A^T \cdot B$)

```
>>> C = tensordot(ones((Nk,Ni)), ones((Nk,Nj)), [0,0])
```

- A 2-way tensor contraction: $C_{IJK} = A_{IJst} B_{sKt}$

```
>>> C = tensordot(ones((Ni,Nj,Ns,Nt)), ones((Ns,Nk,Nt)), [(2,3), (0,2)])
```

Note that the labels on the indices are "dummy" variables. The only requirement is that they match between the sides – just like $f(x, y) = xy$ can be written $f(y, z) = yz$. Also, when an index is summed over, both tensors must have the same size along that dimension (otherwise the sum won't make sense).

Test these out, and check the shapes of the input and output tensors (e.g. `C.shape`) to be sure you understand how the indices are combining. With a little practice, you'll be fluent in the art of index accounting.