Tutorial 1 - Visualizing Linear Algebra

April 29, 2024

1 Introduction

While linear algebra is used in many fields and the geometric interpretation is not always the preferred interpretation of linear algebra[1], in this course we will be focusing mostly on the geometric concepts. To get an intuition of the geometric nature of linear algebra (and its limits), and to further practice working with Blender, in the following assignments we will use Blender to visualize linear algebra exercises. For this tutorial, we'll be assuming that everything is in a 3-dimensional vectors space. Of course, all concepts presented here generalize to any number of dimensions (except for the cross product).

2 Geometry of Linear Algebra

2.1 Vectors

In the previous tutorial we learned how to create meshes from scripts. We can use this to draw the geometric objects that are described by linear algebra. We'll start with the quintessential vector itself.

Exercise: *Write a function `draw_vec` that takes a vector v as a parameter and draws it as an object in the scene.*

Make sure that the parameter v can accept any type of vector, be it a `mathutils` vector, numpy or a python array. Make sure that the vector can be easily seen from any angle. You can draw it in any style you like. Recall that a vector always starts at the origin and points to the value given by v. You can use this function in the remainder of the tutorial when drawing vectors.

Since the arrows require a thickness, you need to offset the geometry of the arrow a little bit to be able to see it. You can start out by just adding a constant value. To do this more beautifully, you should think about the properties of the cross product between two vectors, e.g. what happens when you calculate $\vec{v} \times (0, 0, 1)$?

[1] other interpretations are; polynomials, abstract vector spaces and simple arithmetic

What can you say for certain about the resulting vector? And what happens if you take the cross product again?

To easily distinguish vectors from one another, it is helpful to draw the vectors in a different color each call to the `draw_vec` function. To change the color of the vector we can assign a material to the vector with the following code:

```
# Create a new material
material = bpy.data.materials.new(name="MyMaterial")

# Set the base color of the material
material.diffuse_color = (1, 0, 0, 1)  # these are (R, G, B, Alpha)

obj.data.materials.append(material)
```

Note that materials are assigned to objects and not meshes, the whole object shares the same material. You can change the color by settings its red, green, blue values. The alpha value indicates its transparency, just leave that at 1. It is convenient to assign colors in a particular order. An often used order is red, green, blue, cyan, magenta, yellow, after which you can do arbitrary colors.

Exercise: *You should now be able to visualize the vector cross product. Choose a pair of arbitrary vectors $\vec{v}$ and $\vec{u}$, and draw them and their cross product $\vec{v} \times \vec{u}$ in the scene. You can use the `numpy.cross` function for this.*

Visualizing the dot product is slightly more difficult, as the dot product results in a scalar and not a vector. The dot product is a measure of how "similar" two vectors are. (It is the cosine of the angle between them, scaled by the product of the lengths.) To still get a feel for what the dot product does, we plot it as a multiple of one of the input vectors.

Exercise: *Draw a few arbitrary pairs of vectors $\vec{v}$ and $\vec{u}$ in your scene, together with the vector $d \cdot \vec{v}$ where d is the dot product $\vec{v} \cdot \vec{u}$. Try to get a feel for how the dot product behaves.*

Another important operation on vectors is the vector sum. It can be helpful to visualize what a vector sum actually does.

Exercise: *Write a function `draw_vadd` that takes a list of vectors v_i as parameter and draws each vector connected head to tail.*

Do not draw the resulting vector of the addition in the `draw_vadd` function, if you want to display it, just make an extra call to `draw_vec`.

It can be useful to automatically delete all objects in the scene, instead of doing it manually. Select all objects in the scene by iterating over them, then use `bpy.ops.object.delete()`, consult the Blender documentation for more details.

2.2 Points, Lines and Planes

Vectors are not the only objects that can be described using linear algebra. Points and any form of hyperplane are also important objects of linear algebra. We will start with the regular (3d) plane itself. A plane is usually defined using a *normal* vector, which is a vector that is orthogonal to the plane. (Recall that all hyperplanes are affixed to the origin.)

Exercise: *Create a function `draw_plane` that takes a plane normal n and creates an object in the scene displaying that plane.*

In principle (hyper)planes are infinite, but displaying an infinite plane doesn't help with insight. Define some (global) constants in your script that describe how far "infinite" objects are allowed to extend. Make sure this can be changed easily.

Next is the line. A line in linear algebra is basically the set of points $s \cdot \vec{v}$, where s is some scalar and $\vec{v}$ some vector. It is essentiall all points you can make by scaling the given vector.

Exercise: *Create a function `draw_line` that takes a vector v and creates an object in the scene displaying a line.*

Again you can choose how to display your line, just make sure it visible from any angle. The same goes for the point:

Exercise: *Create a function `draw_pnt` that takes a vector v and creates an object in the scene displaying a point.*

2.3 Matrices

The best way to visualize a matrix is by interpreting its row and column vectors as a basis (three vectors that define the axes of the coordinate system).

Exercise: *Write a function `draw_rowb` that takes a matrix M and draws its rows as separate vectors.*

Again make sure that it doesn't matter what exact type the matrix is. Most of the time this will be a numpy array, but pyhon lists should work as well.

The identity matrix is the matrix that has all zeroes except for the diagonal

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

If you now draw the identity matrix I using this function you will get something

that will look like the standard coordinate axes. This is not a coincidence. Any matrix in essence is a set of coordinate axes, though some axes may overlap.

Exercise: *Write a function* `draw_colb` *that takes a matrix* M *and draws its columns as separate vectors.*

A *linear combination* of two vectors $\vec{v}$ and $\vec{u}$ is a vector that you can get by taking the sum of some scaled version of those vectors, formally a linear combination is a vector $s \cdot \vec{v} + t \cdot \vec{u}$ where s and t are some scalars. Two vectors that *are* a combination are said to be *linearly dependent*. Both these concept extends to any number of vectors, not just two.

Exercise: *Think of an arbitrary* 3×3 *matrix* M*, make sure that* none *of the column vectors are linearly dependent. Draw this matrix using the* `draw_colb` *function.*

This matrix can be thought of as a linear transformation. To transform a vector $\vec{v}$, we typically left multiply $M \cdot \vec{v}$.

Exercise: *Draw the result* r *of the left multiplication* $r = M \cdot \vec{v}$ *in the scene.*

Exercise: *Now using the* `draw_vadd` *function draw the linear combination of the column vectors of* M*. Using the elements of* $\vec{v}$ *as scalars, that is* $x \cdot C_1 + y \cdot C_2 + z \cdot C_3$ *where* $\vec{v} = [x, y, z]$ *and* C_1, C_2, C_3 *the columns of* M*.*

Look closely, what is the pattern here? How was the vector r built? Recall, that a linear transformation is nothing more than changing the basis (the coordinate system).

Exercise: *Now create a new matrix* N*, where the third column is a linear combination of the first two and draw it in the scene using* `draw_colb`*. Also draw a plane using* `draw_plane` *with the cross product of the first two columns (*$C_1 \times C_2$*).*

What do you notice about the vectors and the plane? Why can we say that a plane is a subspace of 3-dimensional space ($\mathbb{R}^3$)?

Exercise: *Draw the row basis for* N *using the* `draw_rowb` *function, over what you drew in the previous exercise.*

What do you notice about the row basis of N? Can you say something about the dimension of the space spanned (formed by taking linear combinations of its basis vectors) by N? We call the dimension that a matrix is able to cover with its basis vectors its *rank*. What can you say about the rank of the space covered by its

columns and its rows?

2.4 Inverse and Gaussian Elimination

When a matrix M is viewed as a linear transformation (by left multiplication), we then call the matrix that undoes this transformation the inverse matrix M^{-1}. We could find the inverse of a given matrix by using the `numpy.linalg.inv` function. But to gain an understanding of what that function does we will first invert a matrix by hand. Lets say we have the following matrix.

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 4 \\ 5 & 6 & 0 \end{bmatrix}$$

And we want to invert this matrix, we can use the following procedure. First write the concatenation (putting matrixes side by side) of M and the identity matrix.

$$\left[\begin{array}{ccc|ccc} 1 & 2 & 3 & 1 & 0 & 0 \\ 0 & 1 & 4 & 0 & 1 & 0 \\ 5 & 6 & 0 & 0 & 0 & 1 \end{array}\right]$$

We know that we can interpret the rows of any matrix as vectors themselves, and thus we can take linear combinations of them. For example we can subtract 2 times the first row $2 \cdot [1\ 2\ 3\ 1\ 0\ 0]$ from the 3rd row, to yield:

$$\left[\begin{array}{ccc|ccc} 1 & 2 & 3 & 1 & 0 & 0 \\ 0 & 1 & 4 & 0 & 1 & 0 \\ 3 & 2 & -6 & -2 & 0 & 1 \end{array}\right]$$

Exercise: *Add and subtract linear combinations of the vectors until the 3×3 submatrix left of the vertical bar is equal to the identity matrix.*

The 3×3 submatrix on the right now contains the inverse M^{-1} of the original matrix. This process is called *Gaussian elimination* (or sometimes row reduction).

Exercise: *Verify in Blender that the matrix you got indeed is the inverse of M.*

Exercise: *Can you explain why this procedure works to invert the matrix? (Think of the matrices as a change in coordinate system.)*

2.5 Solving systems of Equations

It might not seem intuitive, but there is a close connection of matrices to systems of equations. Consider the following set of equations:

$$
\begin{aligned}
x + y + z &= 6 \\
2x - y &= 3 \\
3x - 2y + z &= 4
\end{aligned}
$$

Exercise: *Can you solve this set of equations, and verify that* $(x, y, z) = (\frac{11}{4}, \frac{5}{2}, \frac{3}{4})$.

Now let's take the first equation. it can be read as a condition: the set of points (x, y, z) such that if you add them together they result in 6.

Exercise: *Write code that draws a large number of points using the* `draw_pnt` *function that satisfy the condition* $x + y + z = 6$

What do you notice about this set of points? Can you give a geometric interpretation of the condition $x + y + z = 6$?

Exercise: *Extend the* `draw_plane` *with an extra optional parameter* $\vec{p}$ *that moves the origin of the plane to the point* $\vec{p}$.

You can make an optional parameter in python by adding a default value using the equals sign. Just make sure all the optional values are the last parameters of the function.

Exercise: *Create a function "*`n, p = plane_from_equation(c)`*" that calculates the plane normal* $\vec{n}$ *and origin* $\vec{p}$ *of a plane given coefficients of the form* $c = [a, b, c, d]$ *which satisfies the equation* $a \cdot x + b \cdot y + c \cdot z = d$. *Then use that to draw a plane for the equation* $x + y + z = 6$.

You will notice that there are multiple choices for $\vec{p}$, can you find one that is convenient to use?

Exercise: *Using the* `plane_from_equation` *function, plot all the three equations. Then plot the solution of the equations as a point.*

Can you give a geometric interpretation of solving a system of equations? If there is a geometric interpretation, then there should also be a vector-matrix form for the system. In fact, we can rewrite the above equations in matrix form:

$$
\begin{bmatrix} 1 & 1 & 1 \\ 2 & -1 & 0 \\ 3 & 2 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 6 \\ 3 \\ 4 \end{bmatrix}
$$

If we call the left hand side M and the right hand side $\vec{d}$, and the vector $\vec{x} = [x, y, z]$ we can that equation as:

$$M\vec{x} = \vec{d}$$

Where we want to find a solution for the vector $\vec{x}$. We know that for the matrix inverse we have $M^{-1} \cdot M = I$. Recall that matrix multiplication is non-commutative, it therefore matters whether we multiply from the left or right. If we now multiple the previous equation with M^{-1} on both sides:

$$M^{-1}M\vec{x} = M^{-1}\vec{d}$$

Simplifying:

$$I\vec{x} = M^{-1}\vec{d}$$
$$\vec{x} = M^{-1}\vec{d}$$

This means we can find a solution to the system by finding the inverse of the coefficient matrix and left-multiplying it with the vector of constants $\vec{d}$.

Exercise: *Using the numpy matrix inverse function* `numpy.linalg.inv`*, verify that the system of equations indeed has* $(x, y, z) = (\frac{11}{4}, \frac{5}{2}, \frac{3}{4})$ *as a solution.*

2.6 Minimum Least Squares fit to a Set of Points

The following section is quite a bit more advanced, but if you manage to do this you will have quite an advantage in the coming few assignments.

Say we are given a set of points P (e.g. a point cloud), and want to fit a plane through this set that closely matches that point set. One way to do so is using a least-squares fit. Such a fit minimizes the sum of squared distances from the plane to each point in P. Each point p_i is given by the coordinates x_i, y_i, z_i. The squared distance to some point (also called the error function) is given by:

$$e(a, b, c) = \sum (ax + by + c - z)^2$$

For a plane with equation $ax + by + c = z$. We want to minimize this sum, in essence this means we want to find the values for a, b and c where the error function is minimal. This is often written as:

$$argmin\,(E(a, b, c))$$

Which you should read as, for all the arguments, the argmin function returns the values of the parameters of e where the function e has the lowest value. We can formulate this problem in matrix form:

$$\begin{bmatrix} x_0 & y_0 & 1 \\ x_1 & y_1 & 1 \\ & \cdots & \\ x_n & y_n & 1 \end{bmatrix} \cdot \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} z_0 \\ z_1 \\ \cdots \\ z_n \end{bmatrix}$$

Which we can abstract as $Mx = d$. This system has way more equations than variables, so it doesn't have a "true" inverse. Instead we compute the Moore-Penrose Pseudo-inverse to get a least squares fit approximation, using the following formula:

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} = (M^T M)^{-1} M^T d$$

Exercise: *Write python code that creates a point set P by sampling random values in the vicinity of a plane defined by a normal n fixated at the origin, and plot this point set using the `draw_pnt` function.*

Exercise: *Find a least squares fit to the point set P using the algorithm given above, and plot it into the scene using the `draw_plane` function.*