# 1 Connections: Example Applications of Linear Algebra in Computer Science

## 1.1 How to use this handout

This handout is intended to provide a quick overview, via carefully selected examples, of a subset of Linear Albegra topics that are commonly used in Computer Graphics, Machine Learning, Artificial Intelligence, and Robotics. It is intended to remind you of what you should already be familiar with from your Linear Algebra courses, highlighting the connections between topics in Lin. Alg., and their applications in Computer Science. It is also intended to help you identify what you need to study in-depth: Wherever you find the concepts and examples in this review are unfamiliar and difficult to follow, you should go back to your textbook and study the corresponding chapter(s).

## 1.2 Points and Vectors

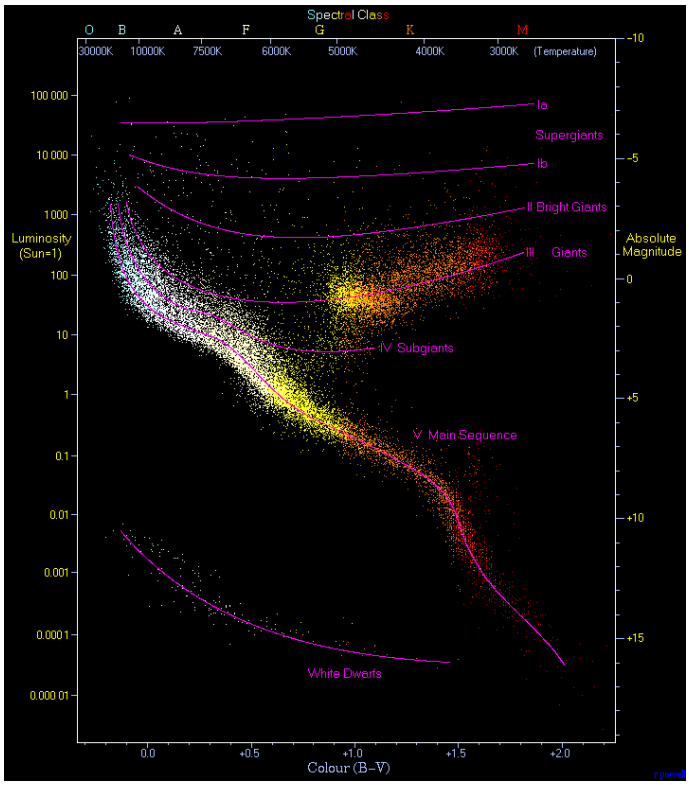
The common thread joining Linear Algebra with many subfields of computer science is its vast power as a tool for analyzing, understanding, and manipulating information. The key for this is the use of *vectors* as a means for representing, storing, and manipulating data.

### 1.2.1 Example

---

The diagram below is a plot of two important properties of stars: Temperature along the $x$ axis, with hotter stars toward the left, and cooler stars toward the right. On the $y$ axis, the plot shows the luminosity (brightness) of the star. These two properties are used to classify stars into a few important groups.

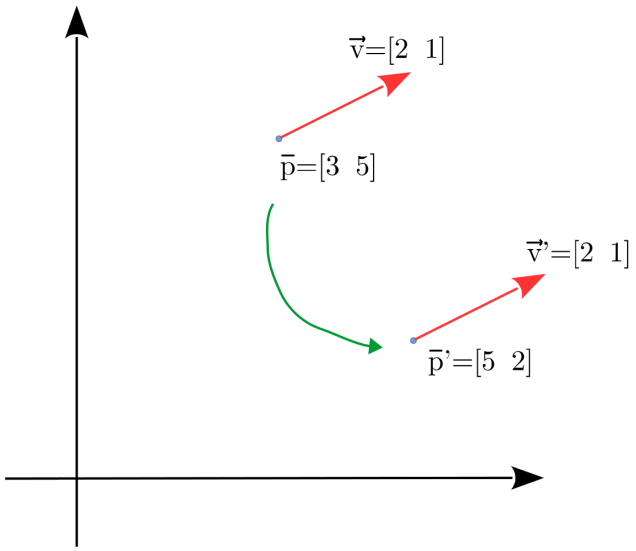



The Hertzsprung-Russell diagram [Source: Wikipedia, author: Richard Powell]

Each star in the plot is represented by two values, so we can create a set of *points* $\bar{p}_i = [t_i \;\; l_i]$ with the coordinates $t_i$ and $l_i$ that give the temperature and luminosity for the corresponding star.

We can also represent each star as vector from the origin of the plot to the star's coordinates $\vec{v}_i = [t_i \quad l_i]$. At first these two may appear to be the same thing. However, there is a subtle but important difference between them. If we change the location of a point, its coorinate values change so we end up with a different point. But we can move a vector's origin around and it remains the same. The vector is only a direction and a length, its component values do not depend on *where* its origin is placed.

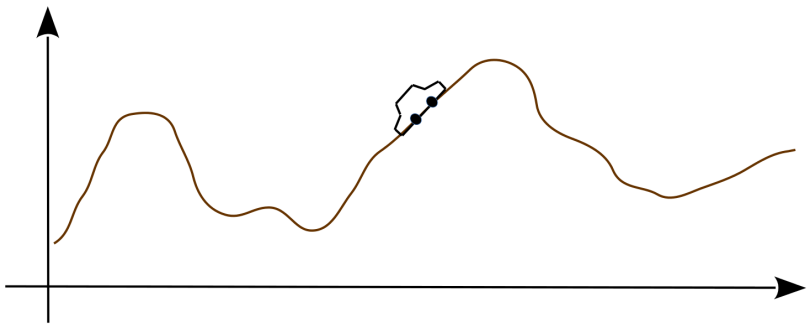



If we displace a point it becomes a different point. If we displace a vector it remains the same

To learn more about the diagram above, click here: `https://en.wikipedia.org/wiki/Stellar_evolution`

---

In A.I. and Machine Learning it is typical to think of data points as vectors from the origin to the point's location. A vector with 3 entries represents a data point in a 3D space, but it doesn't have to be a spatial location. For example, it could represents the position, velocity, and acceleration of a car moving along a road:



A car moving along a mountain road. We wish to represent its position along the $x$ axis, its velocity, and its acceleration

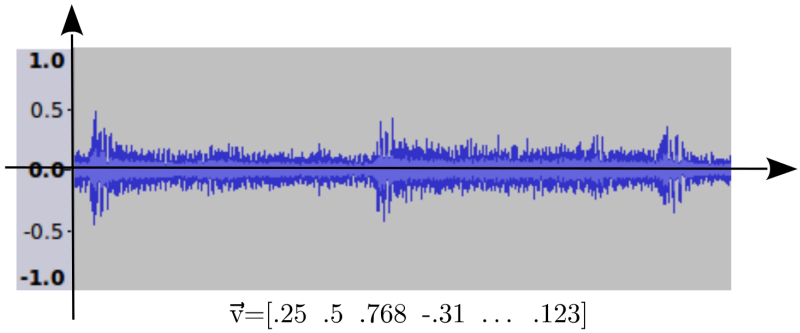
The *feature vector* $\vec{c} = [x_t \quad v_t \quad a_t]$ representing the car at some point in time corresponds to a point in a 3D space of position, velocity and acceleration. Every possible combination of these variables corresponds to some point in this space even though some of these combinations may not be physically feasible for the car to achieve.

Similarly, a feature vector of length $n$ containing values for a set of $n$ features describing some object corresponds to a point in an n-dimensional space. The size of this feature space grows exponentially with the number of dimensions.

This is an important observation. It means that if we are interested in studying the properties of a data set, we can not hope to study the entire feature space our data points belong to, even if the number of features is small. It also means our data set (as large as it may be) will represent only a tiny portion of this feature space. To learn anything meaningful, we need to study the regions in feature space represented in our data set, and then generalize to parts of the feature space for which we have no observations. We will see below a few ways in which Linear Algebra can be used for this purpose.

Examples of data that we can encode using vectors include:

**Sound** (music or speech): Sound data is made of of a sequence of values representing the amplitude (volume) of the sound over time. These values can be stored sequentially into a vector. One second of CD quality music would be represented by a vector with $44,100$ values.

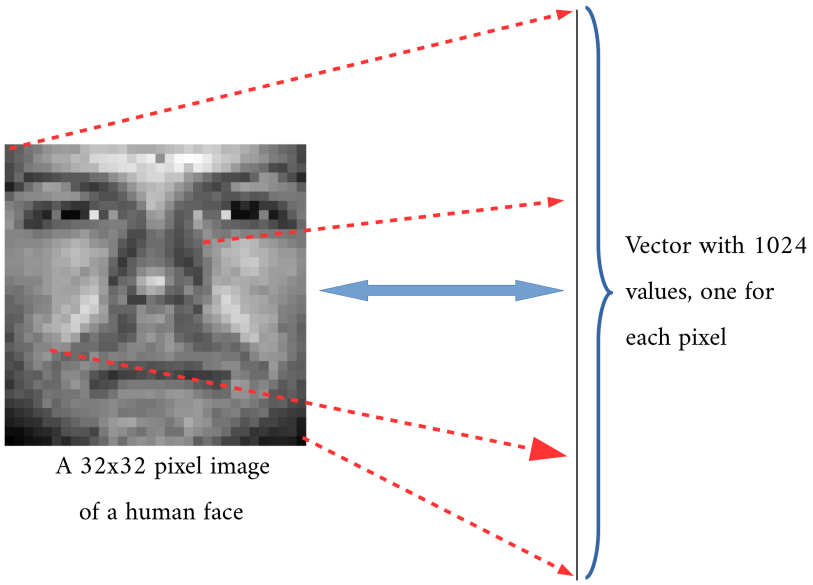



Sound data can be stored in a long vector of amplitude values corresponding to the volume of the sound at each moment in time

**Images** (and video): A single image (or a single frame from a movie) consists of a set of pixels. Each pixel stores either brightness (for grayscale images) or RGB colour data. We can store the data for each pixel into consecutive entries in a vector. If we have an $32 \times 32$ grayscale image such as the picture shown below, we will end up with a vector $1024$ entries long containing the brightness values for each pixel. The conversion works both ways: We can take any 1024 entry

vector and display its values as a $32 \times 32$ image. Thus, every possible $32 \times 32$ image is a point in a $1024$-dimensional space.




An image can be turned into a vector by stacking all the pixels into a long list of grayscale or colour values

**Documents**: There are many possible ways to represent a written document, one of them is to describe the document with a vector where each entry corresponds to one unique word in the document's language (the English language, for instance, uses over $170$-thousand words), and the value stored at each entry is the number of times the word appears in the document.


A visual representation of the word frequency vector for this Linear Algebra review.

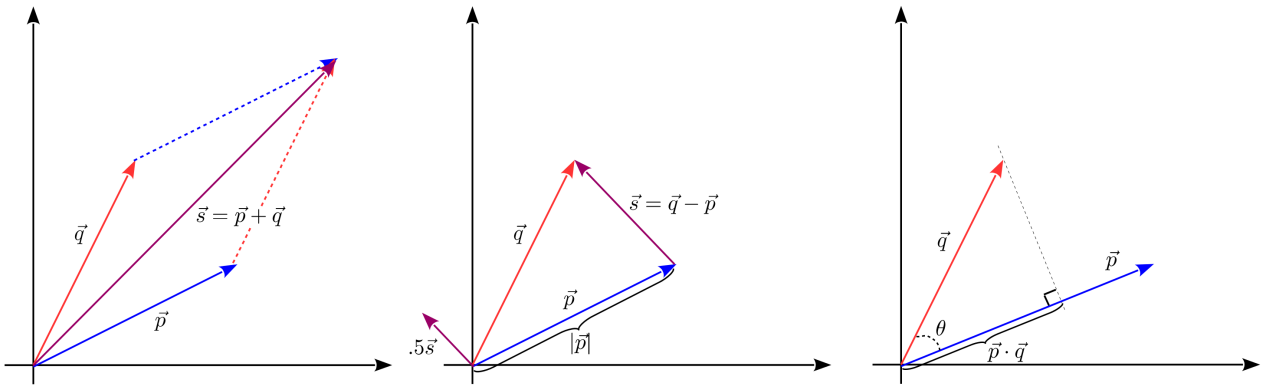
 

With some care, it is possible to devise an appropriate way to encode almost any kind of information, object property, or measurement into a feature vector that will allow us to find useful and meaningful patterns in a data set.

## 1.3 Vector operations

The basic vector operations we are concerned with are:

- **Addition**: $\vec{s} = \vec{p} + \vec{q} = [p_1 + q_1 \quad p_2 + q_2 \quad \dots \quad p_n + q_n]$
- **Subtraction**: $\vec{s} = \vec{q} - \vec{p} = [q_1 - p_1 \quad q_2 - p_2 \quad \dots \quad q_n - p_n]$ (note the order of subtraction)
- **Magnitude**: $|\vec{v}| = \sqrt{\sum_i v_i^2}$
- **Scaling**: $\alpha\vec{p} = [\alpha p_1 \quad \alpha p_2 \quad \dots \quad \alpha p_n]$
- **Dot product**: $\vec{p} \cdot \vec{q} = |\vec{p}||\vec{q}| \cos\theta = \sum_i p_i q_i = pq^T$



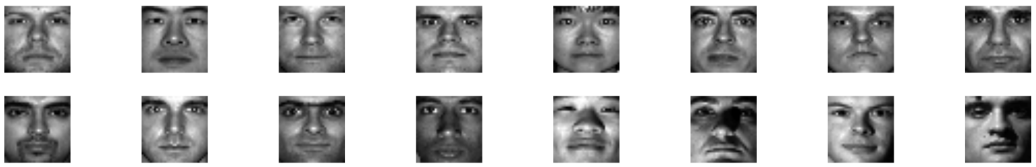

Basic vector operations in 2D

### 1.3.1 example

---

The operations above seem simple enough, but they are already sufficient to let us have some fun with data. Consider a set of pictures of faces. For simplicity, let's assume they are all small (e.g. $32 \times 32$ pixels) and contain only grayscale information.



A small set of pictures of human faces

From the text above, we know we can take each of these faces and transform it into a vector with $1024$ entries. What can we learn from these sample faces using a few simple vector operations? Suppose we compute the following:

$\vec{\mu} = \frac{1}{n}\sum_{i=1}^{n} \vec{f}_i$,

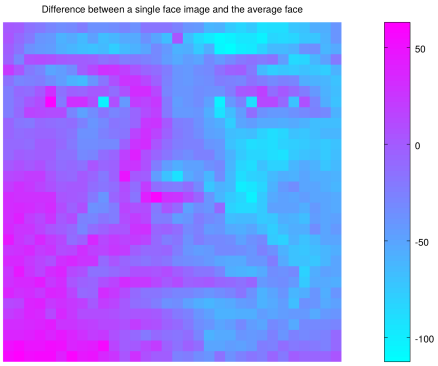
where $\vec{f}_i$ is the $i^{th}$ face vector. The vector $\vec{\mu}$ is the mean or average vector for the dataset. Since it comes from a set of face images, its values should correspond to what an average face looks like at least in out very small data set. Let's reshape the $\vec{\mu}$ vector back into a $32 \times 32$ image and see what that is:


Average face for our small dataset

Does that make sense? keep in mind we used a tiny dataset! given the huge size of the feature space for these vectors, this mean vector probably isn't very good at describing what an average human face really looks like, but it does tell you something about the particular samples we have at hand.

Indeed, the mean vector for a dataset is very often computed as part of much more complex data manipulation or machine learning procedures, we will come back to mean vectors and their applications later on. For now, let's have a look at what happens when we compare the first face in our data set with the mean face vector (and by now it should be clear that we do this by subtracting these two vectors, and re-shaping the result to show it as an image).




Difference between a sample face and the mean face for the dataset

At first glance it may not look like we have learned much. The image looks just like a face with a funny colour. However, if you pay attention to the colorbar at the right of the image, you may notice that the values in the difference image go from somewhere close to -110 to about 60 or so. The range of values is about 170. The range of the original images is 255.

This is meaningful because it suggests we can represent the original images by storing their difference with respect to the mean, along with the mean vector. Why would that be useful? well, the differece images have a smaller range and a smaller amount of internal variation compared to the originals. They contain less information than the originals. That means they can be compressed much more effectively.

For such a tiny dataset it makes little difference, but imagine having millions of different face images: We could save a lot of bandwidth by compressing and transfering the difference images instead of the originals. Of course we also need to transfer the mean face image, but there is only one of those.

This is a very crude way to compress information. As we shall see, there are much smarter ways to achieve even greater compression power while also obtaining much more meaningful information about the structure of our data: The Discrete Cosine Transform (DCT) used in JPEG image compression, and the Discrete Fourier Transform (DFT and its efficient version the FFT) used for sound analysis are examples of simple vector operations that are nevertheless very powerful tools for data analysis.

---

## 1.4 Finding interesting patterns in data

Of the basic vector operations mentioned above, dot products play a particularly important role in analyzing information and finding patterns in datasets. Dot products allow us to measure *distance*

along a specific direction - this is extremely useful for data analysis. They also allow us to measure *alignment* in terms of the angle between vectors - and this is extremely useful in measuring similarity of data items.

To understand how these two kinds of measurements are used, let's look at two examples. First, let's see how we can use dot products to build a very simple Optical Character Recognition (OCR) system for hand-written digits.

### 1.4.1 Example

---

Suppose we are given a dataset consisting of a large number of samples of hand-written digits. Each image has exactly the same size, and we know what digit each image corresponds to - we have been provided with a *label* for the image.

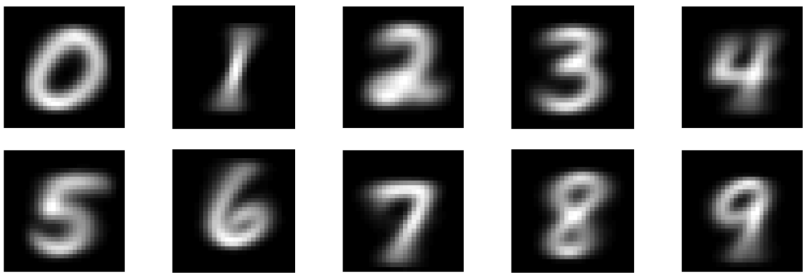
As you have seen above, we can take an image and turn it into a feature vector. For this example, we are using the **MNIST handwritten digits database** (`http://yann.lecun.com/exdb/mnist/`), it contains 10,000 images, of size $28 \times 28$ pixels. Converting them to vectors will result in $10,000$ feature vectors, each with a length of $784$. Importantly, we will normalize each of the feature vectors to be *unit length* by dividing the vector's entries by the vector's original length: $\hat{v} = \vec{v}/|\vec{v}|$.

Our goal is to use this dataset to perform OCR on *new samples* of hand-written digits. We want to be able to *classify* a new $28 \times 28$ image of a hand-written digit that is not part of our dataset and for which we don't know the correct label.

The simplest way to do this is to somehow measure the *similarity* between the new image and those in our dataset. Under the reasonable assumption that images for the same digit are more similar to each other than they are to images of different digits, we can build a simple classifier based on dot products.

To make our classifier really simple and fast, we will compute the average image for each of the digits $0 - 9$ in our dataset. We will use the average image to compare our new one against. In the remainder of the example, keep in mind that all operations are performed on *feature vectors*, even though they are displayed as images in the illustrations to help show what we are doing. Whenever we say "let's take image $x$ and perform operation $y$ on it", we are talking about its feature vector.



Average digits from 10,000 handwritten digit samples

Before we proceed, it is important to understand *why* it is that dot products are useful for measuring similarity. Recall the definition of a dot product $\vec{p} \cdot \vec{q} = |\vec{p}||\vec{q}| \cos(theta)$. When both $\vec{p}$ and $\vec{q}$ are unit vectors the dot product reduces to the angle $\theta$ between them. The dot product in this case is telling us how well the two vectors are aligned with each other. This is interesting because as it turns out, *the direction of a vector does not change much with small changes to the values of its entries*.

Put in a different way. If we have two samples of a digit, say $3$, and they look similar to each other, we can expect their feature vectors to have a similar direction, and the corresponding dot product will be close to $1$. Conversely, the feature vectors for different digits should be expected to point in reasonably different directions, and a dot product between them will result in a value significantly smaller than $1$.

To show this, let's see what happens when we take the average image for a $3$, and distort it in different ways. The simplest thing to do is to change its brightness by scaling the image by some constant factor. Changing the length of a vector does not affect direction, so once we take the modified image's feature vector and normalize them to unit length the dot product with the original remains at $1$. What happens if we add a small value to each entry in the image's feature vector (not the same as scaling!), or when we distort the image by adding a scratch to it is more interesting. In both cases you can see the dot product with the original image remains large. Finally, if we compare the average $3$ with the average $1$ we see that the dot product is much smamller.

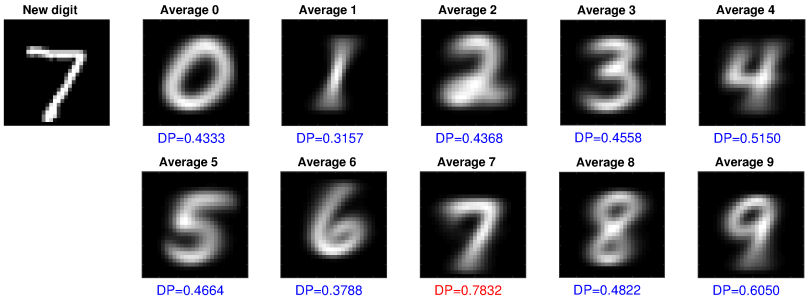

Comparison between the average $3$ and distorted versions of itself, as well as with the average $1$

The important message here is that vector direction is stable to changes in the feature vector that preserve the pattern of the information encoded by it. We will use this property to build our

classifier. Having computed the average images for each digit, the simplest classifier is trivial to implement:

- Obtain a unit length feature vector for the new digit image
- Compute the dot product of the new digit with each average digit image
- Select the label corresponding to the average image with the largest dot product




Classifying a new handwritten digit image of a 7. The dot product of this new image with each average digit is shown. As expected the correct one has the largest dot product.

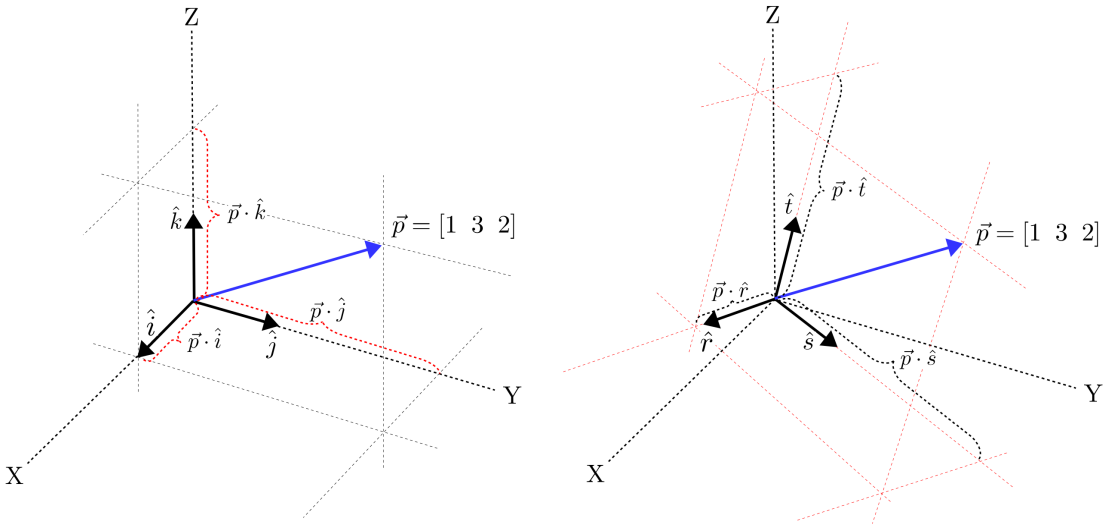
This is the simplest form of a *nearest neighbour* classifier. However, it still manages to achieve an $82\%$ correct classification rate on $10,000$ test images not part of the dataset we used to compute the average digits. Quite impressive for such a simple classifier. Of course, there are better ways to build nearest neighbour classifiers, and different ways to measure similarity of vectors. You will get to learn about all of these in your Machine Learning course.

---

Beyond machine learning and A.I., dot products between unit-length vectors find extensive use in computer graphics, where they are used to specify the orientation of object surfaces, to determine what happens to light rays as they get reflected or refracted by objects in a scene, and to determine the colour of an object given the object's material properties and the direction and intensity of light shining on it.

The second common use of dot products is that of measuring distances along specific directions. The figure below shows a vector $\vec{p}$ in 3D space. The left diagram shows the standard coordinate axes with their associated unit-length vectors $\hat{i} = [1 \;\; 0 \;\; 0]$, $\hat{j} = [0 \;\; 1 \;\; 0]$, and $\hat{k} = [0 \;\; 0 \;\; 1]$. We can easily break down vector $\vec{p}$ into its components along each of the coordinate axes by taking the dot product of $\vec{p}$ with the corresponding unit vector. Because the original vector's coordinates are in the $X$, $Y$, $Z$ coordinate frame, we simply get $1$, $3$, and $2$ for the components along each axis.




Dot products are used to break down a vector $\vec{p}$ into components along a set of orthogonal directions.

This is not very interesting - what is interesting, is that we can perform the same kind of measurement on a completely different coordinate frame as long as

- The coordinate frame consists vectors that are mutually orthogonal
- The vectors are all unit-length

Such a coordinate frame is called *orthonormal*, and the vectors that form the coordinate frame are called basis vectors. The right side of the figure shows an example of such an orthonormal coordinate frame formed by the unit vectors $\hat{r}$, $\hat{s}$, and $\hat{t}$. Once again we can break down vector $\vec{p}$ into components along each of the axes of the new coordinate frame. In this particular case, we will obtain three scalar values $p_r = \vec{p}\cdot\hat{r}$, $p_s = \vec{p}\cdot\hat{s}$, and $p_t = \vec{p}\cdot\hat{t}$.

There are several important observations to be made here. First, given an orthonormal basis for some coordinate frame, the projection of $\vec{p}$ onto each of the basis vectors *encodes* all the information represented by $\vec{p}$. In this case, $\vec{p}$ corresponds to a position in space, so the previous statement says that for the coordinate frame of $\hat{r}$, $\hat{s}$, and $\hat{t}$, the coefficients $p_r$, $p_s$, and $p_t$ encode exactly the original location of the point at the tip of $\vec{p}$. Another way to think about this is that we can *reconstruct* or *recover* the original vector $\vec{p}$ from its components along each of the directions that constitute the orthonormal frame: $\vec{p} = p_r\hat{r} + p_s\hat{s} + p_t\hat{t}$.

Secondly, there is no redundancy or correlation between the components of the original vector along each of the directions that form our orthonormal frame. This is important. Each basis vector captures information from $\vec{p}$ that is completely de-correlated with what is captured by any other basis vector.

To truly appreciate the usefulness of breaking down a vector into a set of components along particular directions, we need to look at what happens when the vectors we are manipulating represent more complex information. Let's have a look at how we can use the principle described above to analyze an audio clip.

### 1.4.2 Example

---

Consider the short audio clip shown in the image below.

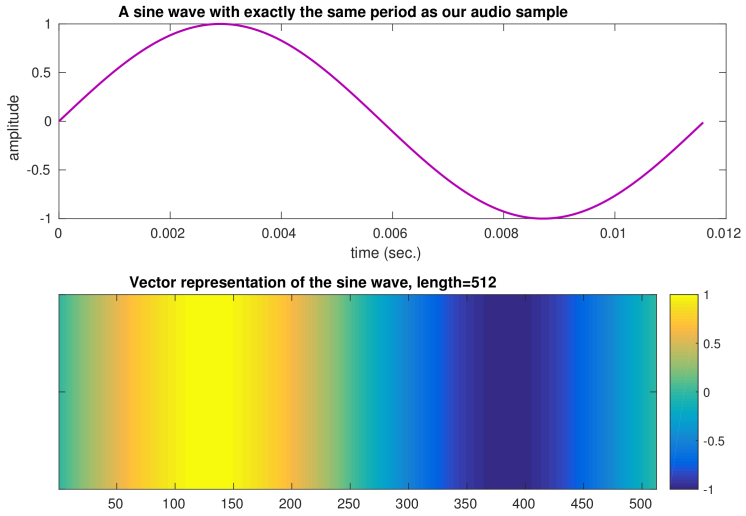

A short audio clip from a song - at CD quality this sample would contain 512 amplitude measurements.

As noted above, we can take the sampled values that make up this clip, and store them in a vector. In this case the vector will have 512 elements, each corresponding to the amplitude of the sound at succeeding time points evenly spaced between $0$ and the end of the clip.

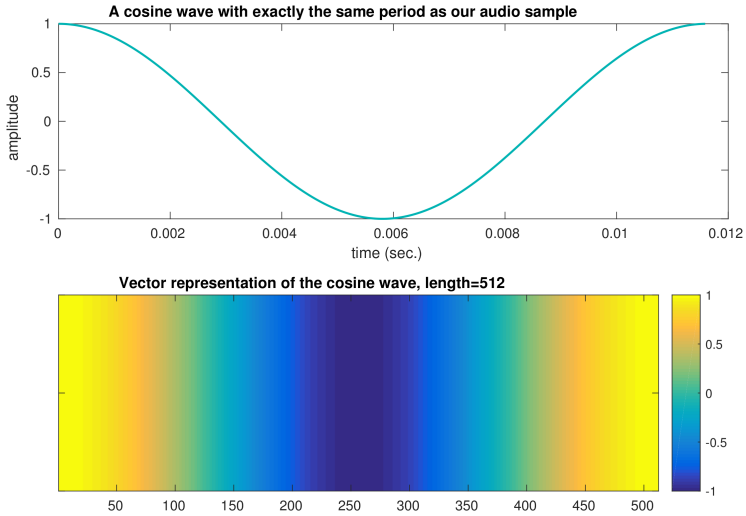
The resulting 512-entry vector $\vec{a}$ represents a point in a 512-dimensional space. We can build an orthonormal coordinate frame for this space using 512 basis vectors that are unit length, and that are mutually orthogonal. We could pick any set of basis vectors that satisfies these conditions. The key idea here is that *we can choose the direction of our basis vectors so that it tells us something interesting about the signal*.

One of the most useful choices of directions for basis vectors used in signal analysis corresponds to sine and cosine waves of various frequencies as shown below.




A sine wave whose period is exactly the same as the duration of our input clip. It is sampled to produce a 512-entry vector representing this specific sinusoidal function.




The corresponding cosine wave and the corresponding 512-entry vector representation.

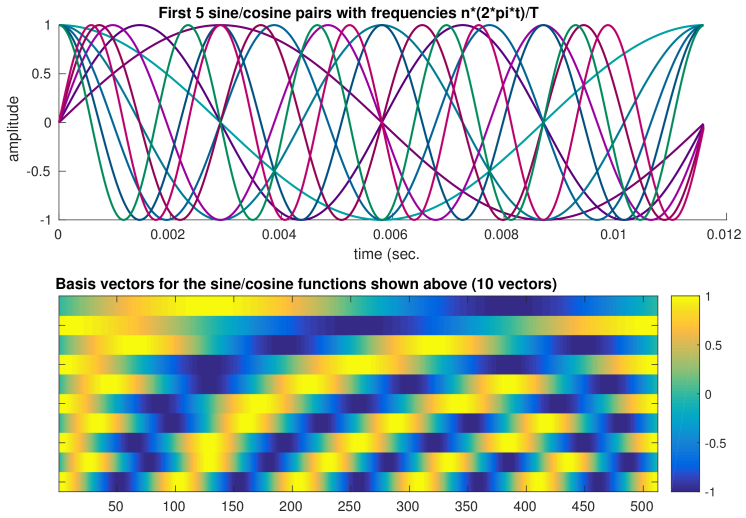
Let's call the frequency of these two sinusoidal waves $f_1$. It corresponds to the lowest frequency sinusoid that can be fully contained in a sample of the same duration as our input. Let's call the vectors from the two waves shown above $\vec{s_1}$ and $\vec{c_1}$, and normalize them to obtain unit vectors $\hat{s_1}$ and $\hat{c_1}$. These two unit vectors are the first two directions in our orthonormal basis.

Note that by definition, $\hat{s_1}$ and $\hat{c_1}$ are orthogonal: sine and cosine are $90$ degrees out of phase with each other, so $\hat{s_1} \cdot \hat{c_1} = 0$. To build the rest of our orthonormal basis, we make use of the

observation that sine and cosine waves whose frequencies are integer multiples of each other are mutually orthogonal. So, if we create a sine wave with $f_2 = 2f_1$, sample it, and normalize it to obtain unit-length vector $\hat{s_2}$, we will observe that $\hat{s_2} \cdot \hat{s_1} = 0$ and $\hat{s_2} \cdot \hat{c_1} = 0$. Similarly, we obtain a basis vector $\hat{c_2}$ by sampling and normalizing a cosine wave with frequency $f_2$. It can be readily verified that its dot product with any of the remaining three vectors in our set yields $0$.

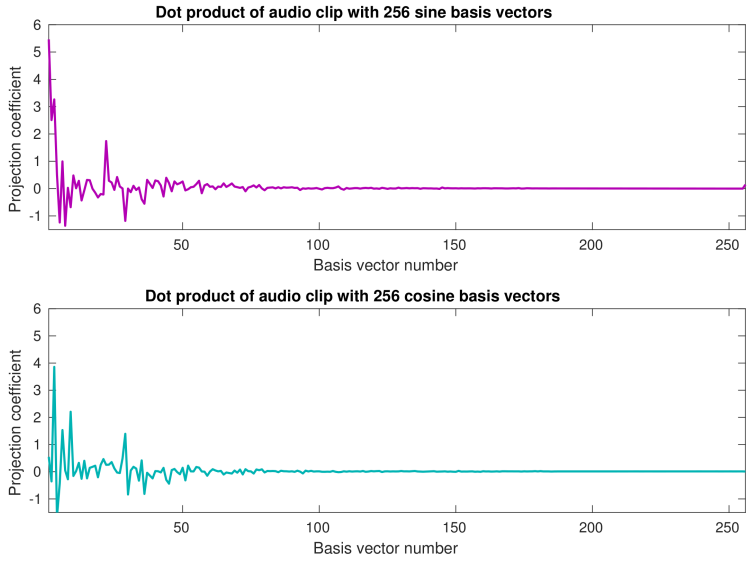
So, to build our orthonormal basis, we generate pairs of sine/cosine waves with frequencies $f_1, 2f_1, 3f_2, \ldots, 256f_1$, then sample and normalize these waves to obtain a total of $512$ basis vectors. The first few sine and cosine waves, along with their corresponding sampled vectors are shown below.




The first 5 sine and cosine waves that are used to build our orthonormal basis set, along with the corresponding vectors (before normalization).

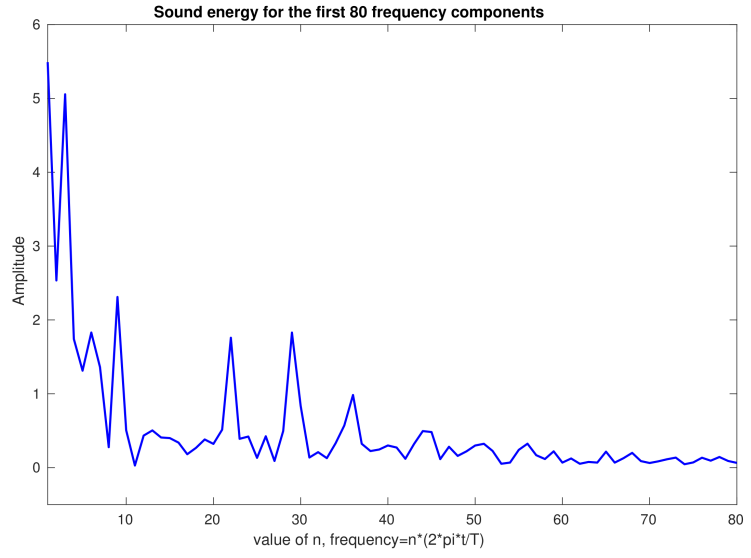
Given our full orthonormal basis, we can break down our audio sample into components corresponding to each of the frequencies represented by the sine and cosine waves in our basis vectors. The plot below shows the value of the components resulting from taking the dot product of $\vec{a}$ with each of the basis vectors in our orthonormal basis - grouped by type (sine or cosine), and in order of increasing frequency. The component corresponding to $f = f_1$ is left-most in the plot, and the component for $f = 256f_1$ is at the right-most position.




Frequency content of our original audio clip in terms of sine and cosine waves of increasing frequency.

From these plots, we can quickly see which frequency components are present in our clip, and in what amount. Indeed, it is clear that our clip doesn't have a lot of energy in high frequency components - most of the components for higher frequencies are zero or close enough to zero that we can ignore them. To gain a better idea of how much *energy* in the original clip comes from each of the different frequencies, we take corresponding sine and cosine components and compute energy as $E = \sqrt{(x_k^2 + y_k^2)}$ where $x_k$ is the coefficient for the $k^{th}$ sine basis vector, and $y_k$ is the coefficient for the $k^{th}$ cosine basis vector. The image below shows the energy content for the first $80$ frequency components.



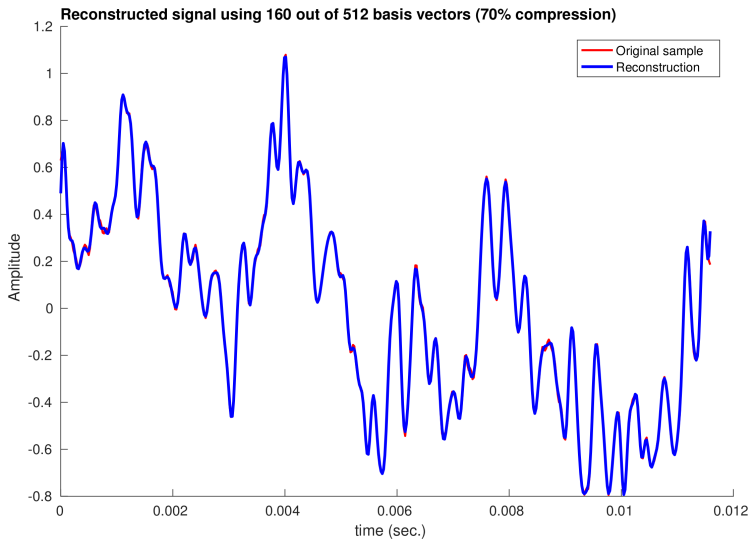

 

Sound energy at each of the first 80 frequency components.

It is easy to see that most of the sound energy comes from the lower frequencies, with energy following a general decreasing trend toward the right side of the plot.

What we have done so far deserves a bit of thought. We have taken an audio clip, represented it as a vector ina 512-dimensional space, and then decomposed it into components corresponding to single frequency sinusoidal waves. The resulting coefficients provide us with information of the frequency content of the clip - in effect this tells us what the audio clip is made out of in terms of single frequency sine and cosine waves!

This has a number of applications. First, remember from the 3D point example that the coefficients we obtain from projecting our input clip onto an orthonormal basis completely represent the clip itself. We can reconstruct the original clip by multiplying each basis vector by the corresponding coefficient, and adding everything up: $\vec{a} = \sum_k \; x_k \hat{s_k} + y_k \hat{c_k}$. However, we can do a lot more. As noted above, most of the higher frequency components in our clip have coefficients close to zero, this suggests we can compress the audio clip by storing only the lowe-frequency coefficients, and using only those coefficients along with the corresponding basis vectors to reconstruct the clip. The process is *lossy*, the reconstructed clip will not be identical to the original, but because the frequencies we are dropping contain almost no energy, the difference will be very small. The plot below shows the result of using only the lowest 80 frequencies (both sine and cosine coefficients for a total of 160 values) to reconstruct the audio clip.




Reconstruction of the original clip using only the lowest 80 frequency components.

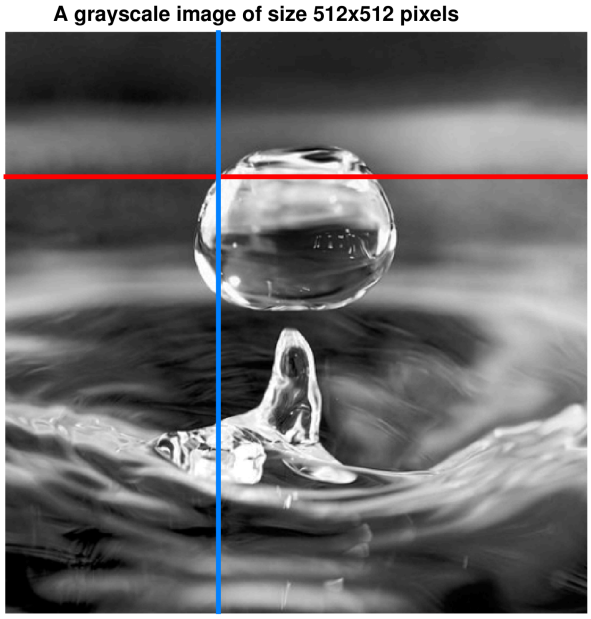
The difference between the original clip and the reconstructed one is negligible, and at the same time, we are able to represent a clip that contains $512$ values using only $160$ coefficients using only $30\%$ as much storage space!

Furthermore, we could change the amount of each component frequency by scaling the appropriate coefficients before reconstructing the signal. We can boost or diminish individual component frequencies at will, effectively achieving the same effect as an audio equalizer.

To summarize: Vector dot products allow us to analyze information by breaking it down into components that are meaningful. The specific form of the orthonormal basis used to analyze our data will depend on the type of data we are working on, and what we intend to use the resulting coefficients for. There are many kinds of orthonormal basis functions commonly used in signal analysis, but the principle is the same, and they each find applications in signal processing (analysis, filtering, re-shaping), information compression, pre-processing for machine learning or AI, and information visualization.

The process above is a very simplified form of one of the most common techniques for signal analysis: The Discrete Fourier Transform. Note that the DFT does not use separate sine/cosine waves as a basis, instead, it uses complex exponential functions, but remember that Euler's identity states that $e^{i\theta} = cos(\theta) + isin(\theta)$ so in effect it is decomposing the signal into sinusoidal frequency components just as we have done above. The DFT has many applications, from audio processing to speech recognition to solving partial differential equations.

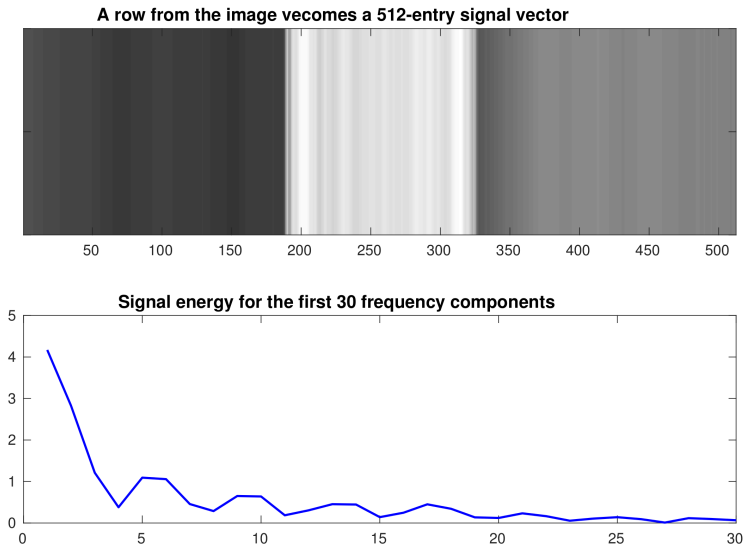
Beyond audio, frequency analysis can be applied to any input vector that contains values from a time-varying, or space-varying signal. As a final example, consider the image below.



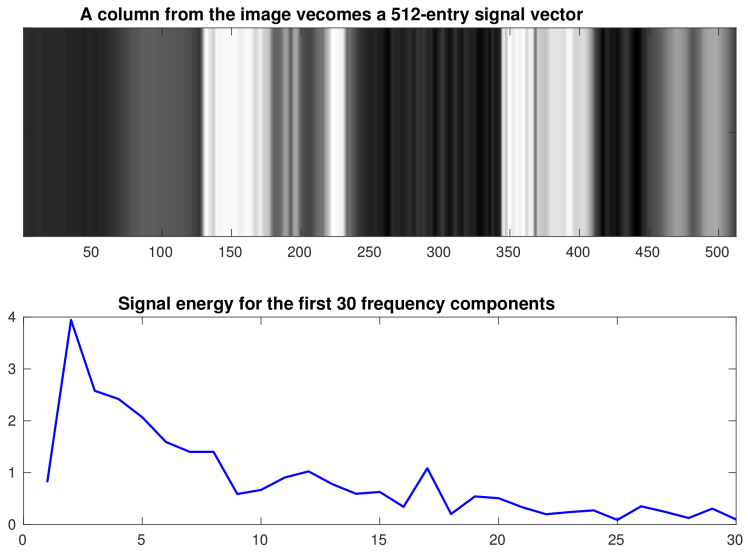
An input grayscale image is a space-varying signal. A row or column in this image corresponds to brightness values along a specific direction and at a specific location over the image. [Original source: Wikipedia, author: Jose Manuel Suarez]

We can take a row from this image, put it into a vector representing a space-varying signal (in this case brightness), and break it down in the same way we did our audio sample before. This will give us information about the frequency content of the image at that specific row along the horizontal direction. Similarly, we can take a column, put it in a vector, and perform our frequency analysis on that to obtain information about frequency content at that specific column along the vertical direction.



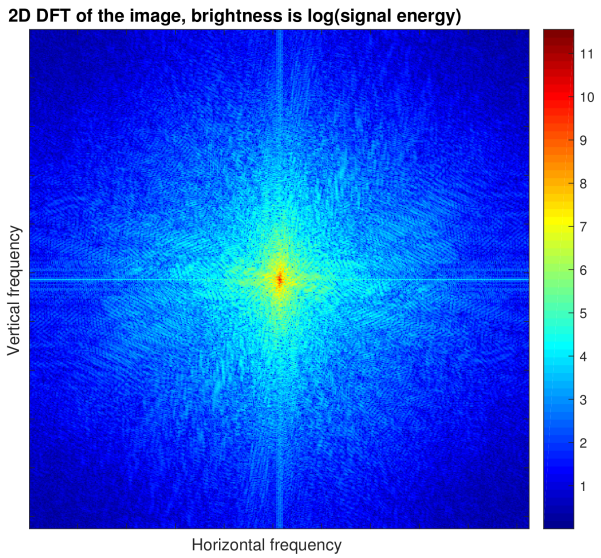

Frequency analysis of one row from the input image




Frequency analysis of one column from the input image

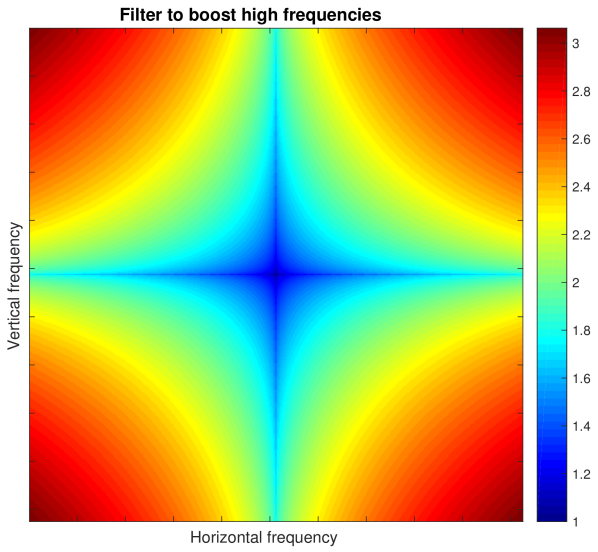
In general, we can sample the image along a line with arbitrary orientation, placing the measured brightness values in a vector, and then we can perform our frequency analysis to determine the frequency content of the image along a specific spatial direction. We can use the 2D DFT to determine the frequency content of the image along all possible orientations. The result is a 2D map with the origin at the center, and with frequency increasing with distance from the origin. Each point in the plot shows the energy of the signal at a given frequency and orientation.




Frequency content of the input image. Each point shows signal energy at a specific location and frequency

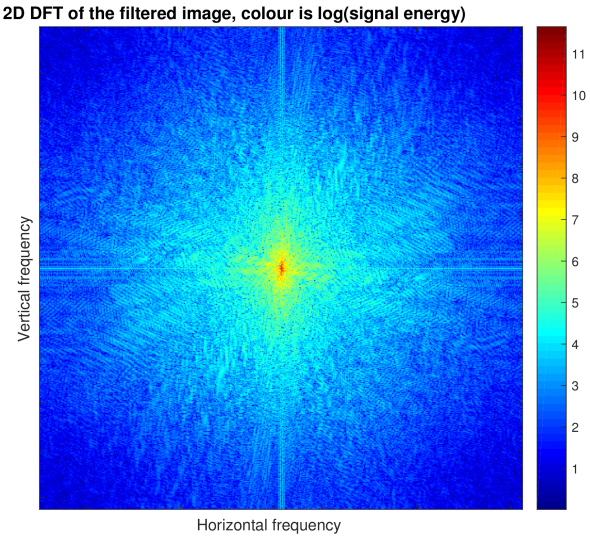
 

As in the audio sample above we can see that, in general, signal energy decreases as frequency increases, with the lower frequencies concentrating most of the signal's energy. Just like with the audio sample we could use this for compression. Instead, let's see what happens if we manipulate the energy content of the signal. In this case, let's boost the high-frequency content by applying the filter mask shown below.




High-frequency boosting filter mask.

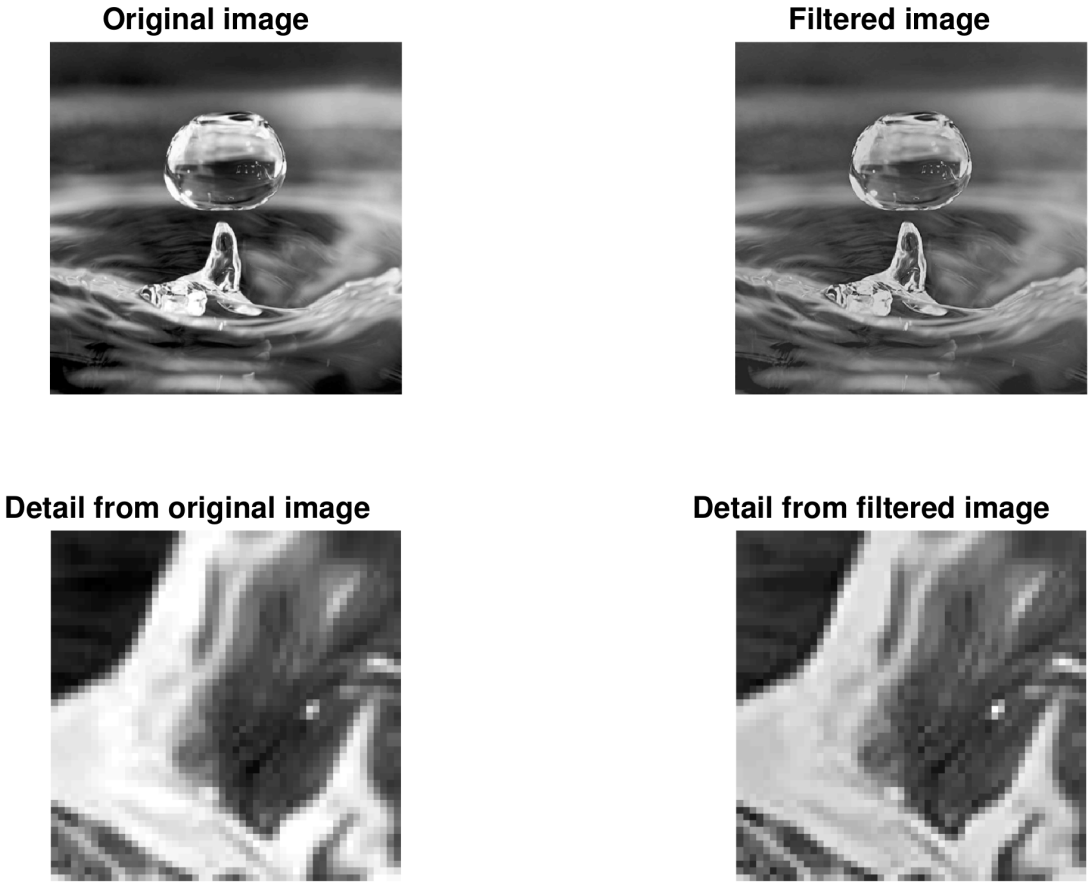
We apply the filter above by multiplying the coefficients for each frequency/orientation by the corresponding value at the same position in the filter mask. The resulting energy plot is shown below



Frequency content of the image after boosting high-frequency components.

Comparing that with the original frequency map, we can see the high frequencies have larger magnitudes overall. We can use these modified coefficients to reconstruct the image. Since we have artificially amplified the high frequency content, and in terms of image structure that corresponds to increasing the amplitude of fast changes in brightness, we expect the filter above to have an *image sharpening* effect.




Original image and sharpened version reconstructed from the modified coefficients.

Comparing the original image with the reconstruction, we can see in the fine detail that there is indeed a sharpening effect. The effect is subtle, but shows the kind of manipulation that is easily achieved by having access to frequency information.

The examples above should have allowed you to understand how you can use vectors and vector operations to represent, manipulate, and analyze information in fairly sophisticated ways. However, we have only started looking at what Linear Algebra can do. Let us move on now to matrices, and look at what they do for us in terms of data representation, manipulation, and analysis.

## 1.5 Matrices and Matrix Operations