AMAT 503: Wavelets and Signal Processing

Michael P. Lamoureux

November 20, 2017

Contents

0 Introduction

These notes follow the content of the course AMAT 503 at the University of Calgary. They focus on the theory of wavelets and their application to signal processing – in particular to 1D signals (such as sound) and 2D signal (such as photos). For a more in-depth analysis, please read the course textbook (which this year is Patrick van Fllet's "Discrete wavelet Transformations.")

1 Lecture 1: Why wavelets

What is a wavelet? It is a little wave!

We use them to decompose complicated signals or images into simpler parts – usually a sum of various wavelets with nice properties (Haar wavelets, Morley wavelets, Daubechies wavelets, etc).

Mainly interested in applications – how do we compress or process digital signals, such as audio, music, seismic traces, digital photos, etc?

There is also a very well-developed mathematical theory that goes with wavelets. We will touch on some of that as well in this course.

Key advantages of wavelet are that they give fast, efficient algorithms for working with real data, and can provide physically intuitive interpretation of the signals under study.

2 Lecture 2: Vector spaces and their properties

2.1 Review

We did a quick review in class of vector spaces; these notes are just a summary. Recall a vector space is a collection of "things" that we can add and subtract, and multiply by scalars (which are real numbers in this class. Sometimes we might want to use complex numbers.) Obvious examples are n-dimensional Euclidean space $\mathbb{R}^n$ like we see in linear algebra, but also include things like the space of continuous functions on an interval, say $C[-1, 1]$, or the set of differentiable functions $C^\infty[-1, 1]$.

You should recall linear transformations, matrices, eigenvalues, eigenvectors, and a few other familiar concepts from your linear methods courses.

2.2 Inner products

We will use inner products a lot, so you should be familiar with them. Given two vectors u, v, we write their inner product using angle brackets: $\langle u, v\rangle$. There are many choices for inner products in a vector space, although some are more natural than others. In Euclidean space $\mathbb{R}^3$, for instance, the inner product is the usual dot product, expressed as the sum of the product of the components:

$$\langle (x, y, z), (x', y', z')\rangle = xx' + yy' + zz'. \tag{1}$$

But you could also have a weighted inner product, say

$$\langle (x, y, z), (x', y', z')\rangle = 2xx' + 3yy' + 5zz'. \tag{2}$$

For functions on a space, often the inner product is given by an integral: on $C[-1, 1]$, the inner product of two functions can be defined as the integral

$$\langle f, g\rangle = \int_{-1}^{1} f(x)g(x)\, dx. \tag{3}$$

Although, again, choices are involved, so someone might like to insert a weighting function, and define a different inner product, like

$$\langle f, g\rangle = \int_{-1}^{1} f(x)g(x)x^4\, dx. \tag{4}$$

To be a proper inner product, the function $\langle\cdot, \cdot\rangle$ must be real valued, positive, symmetric and bilinear. More precisely,

- $\langle u, v\rangle$ must be a real number, for any vectors u, v in the space under consideration;
- $\langle u, u\rangle$ is positive, for all non-zero vectors u
- $\langle u, v\rangle = \langle v, u\rangle$ for all vectors in the vector space
- $\langle u_1 + \lambda u_2, v\rangle = \langle u, v\rangle + \lambda\langle u_2, v\rangle$ for all vectors u_1, u_2, v and real numbers λ. That is, it is linear in the first argument (and in the second argument by symmetry).

From these 4 simply properties, it turns out that the inner product of any two vectors is always smaller than a certain other product. We can state it as a theorem:

Theorem 1 (Cauchy-Schwartz inequality) *Given two vectors u, v in an inner product space, then it is alway true that*

$$\langle u, v\rangle \leq \sqrt{\langle u, u\rangle}\sqrt{\langle v, v\rangle}. \tag{5}$$

Moreover, equality is obtained if and only if the two vectors are parallel (or anti-parallel).

This result is ridiculously easy to prove. Just write down this polynomial:

$$p(t) = \langle u + tv, u + tv\rangle \geq 0, \tag{6}$$

check that it is a quadratic, and use the quadratic formula to obtain the inequality.

The quantity $\sqrt{\langle u, u\rangle}$ is called the length of the vector, and is denoted $\|u\| = \sqrt{\langle u, u\rangle}$. In Euclidean space $\mathbb{R}^3$, with the usual inner product, it corresponds to our usual geometric measure of length. For functions, it is often interpreted as something like energy, or more precisely, root mean square of amplitude.

With this notation for length, the Cauchy-Schwartz inequality can be rewritten as

$$|\langle u, v\rangle| \leq \|u\| \cdot \|v\|. \tag{7}$$

With this in mind, we can define the angle θ between any two vectors by the formula

$$\cos\theta = \frac{\langle u, v\rangle}{\|u\| \cdot \|v\|}, \tag{8}$$

since this ratio is always between -1 and 1. IN Euclidean space, this corresponds to our usually geometric idea of angle in three dimensions. In higher dimensions, and on function spaces, it is a measure of how similar the vectors (function) are – it is a correlation.

Two vectors are perpendicular, or orthogonal, if their inner product is zero. Again, in Euclidean space, this means exactly what you think it means.

2.3 Orthonormal vectors, and Gram-Schmidt

It is very useful to create sets of vectors (or functions) that are orthogonal to each other, and each having unit length. We will be interested in orthogonal wavelets, so let's review how we do this.

The basic idea is to subtract the projection of one vector onto another, to obtain something that is orthogonal. Then adjust the length.

Gram-Schmidt process:

Given a set of vectors $u_1, u_2, \ldots, u_n$, we will create a set of orthonormal vectors $v_1, v_2, \ldots v_m$, that span the same subset as the $u's$.[1]

First, start with u_1. If it is zero, throw it away and go to the next vector. If it is not zero, set

$$v_1 = \frac{u_1}{\|u_1\|}. \tag{9}$$

Next, take u_2 and subtract its projection onto v_1, as follows:

$$u_2 - \langle u_2, v_1 \rangle v_1. \tag{10}$$

If this result is zero, throw it away and go to the next vector. If it is not zero, then adjust it to length one, and call that v_2:

$$v_2 = \frac{1}{\|u_2 - \langle u_2, v_1 \rangle v_1\|} \left(u_2 - \langle u_2, v_1 \rangle v_1 \right). \tag{11}$$

Continue in this way, subtracting the projections onto all previous $v's$. So, for instance,

$$v_k = \frac{1}{\| \ldots \|} \left(u_k - \langle u_k, v_1 \rangle v_1 - \langle u_k, v_2 \rangle v_2 - \ldots - \langle u_k, v_{k-1} \rangle v_{k-1} \right). \tag{12}$$

It is easy to check that the resulting vectors v's are perpendicular to each other, with unit length, and span the same set as the u's.

This works with functions just as well as with n-dimensional vectors.

As an example, take the three functions $1, x, x^2$ in the space of continuous functions $C[0,1]$, usual integral for the inner product, and compute the corresponding orthonormal functions. (We did this in class. It's pretty easy, just tedious.)

2.4 Other lengths, or norms

We saw above that when we have an inner product, the length (or norm) of a vector is given as the square root

$$\|u\| = \sqrt{\langle u, u \rangle}. \tag{13}$$

[1]Same span means: all linear combinations of the u's can be written as linear combinations of the v's, and vice versa.

There are other ways to define a norm on a vector space. Very common are the p-norms: in n-dimensional real space $\mathbb{R}^n$, we define it via the p-th root of the sum of p-powers of the components in the vector, so for a vector $u = (u_1, u_2, \ldots, u_n)$, we write

$$\|u\| = \|u\|_p = (|u_1|^p + |u_2|^p + \ldots + |u_n|^p)^{1/p} . \tag{14}$$

For any fixed number p between one and infinity, this length satisfies the triangle inequality,

$$\|u + v\| \leq \|u\| + \|v\|, \qquad \text{for any vectors } u, v. \tag{15}$$

For p smaller than one, the triangle inequality fails, so we don't call that a length – it is too weird to be useful.

We can also take the limit as p goes to infinity, to obtain the sup norm:

$$\|u\| = \|u\|_\infty = \max(|u_1|, |u_2|, \ldots, |u_n|). \tag{16}$$

All these norms have their usefulness in certain applications.

For function spaces, we can define similar norms using integrals: e.g.

$$\|f\| = \|f\|_p = \left(\int |f(x)|^p \, dx \right)^{1/p} . \tag{17}$$

Again, we get the triangle inequality for p between 1 and infinity. Note that $p = 2$ is the usual inner product-based length or norm.

3 Lecture 3: Numerical matrix methods

3.1 Cholesky factorization

The Gram-Schmidt process described above is sort of a pain to do by hand, and even if you implement it in computer code (e.g. MATLAB), the algorithm as presented is numerically unstable. Which is to say, small numerical errors accumulate and eventually give bad answers in the calculations.

A better way is to make use of robust numerical algorithms that are well-known and easily accessible from any standard numerical math software. For Gram-Schmidt, we can use Cholesky decomposition of matrices.

Cholesky factorization takes a matrix A and writes it in the form of a lower triangular matrix L times its transpose:

$$A = LL^*, \tag{18}$$

where the matrix L has non-zero entries only on the triangle of entries on or below the main diagonal. Moreover, we can insist that L have strictly positive entries on its diagonal. Obviously, not every matrix can be expressed in this form. For, if matrix A is in this form, $A = LL^*$, then it must be square (i.e. n rows and n columns), it must be symmetric (equals its own transpose) and it must be positive (all its eigenvalues are positive – since LL^* has positive eigenvalues). But that's all you need.

Theorem 2 (Cholesky decomposition) *Suppose A is a positive, symmetric, $n\times n$ matrix. Then there is a lower triangular $n \times n$ matrix L with positive entries on the diagonal so that*

$$A = LL^*. \tag{19}$$

It's not hard to create an algorithm to find L. For instance, suppose you have a 3x3 matrix

$$A = \begin{bmatrix} 1 & 2 & 4 \\ 2 & 13 & 23 \\ 4 & 23 & 77 \end{bmatrix}. \tag{20}$$

To find the corresponding L you just solve the system of equations given by

$$\begin{bmatrix} 1 & 2 & 4 \\ 2 & 13 & 23 \\ 4 & 23 & 77 \end{bmatrix} = \begin{bmatrix} a & 0 & 0 \\ b & c & 0 \\ d & e & f \end{bmatrix} \begin{bmatrix} a & b & d \\ 0 & c & e \\ 0 & 0 & f \end{bmatrix} = LL^*. \tag{21}$$

Multiplying out the first row of L with the first column of L^* we get

$$1 = a^2 \tag{22}$$

from which we conclude $a = 1$. Multiplying the second row of L with the first column of L^* we get

$$2 = ab = 1b \tag{23}$$

from which we conclude $b = 2$.

Next we multiply the second row with the second column to get

$$13 = b^2 + c^2 = 4 + c^2 \tag{24}$$

from which we conclude that $c = 3$. And so on, eventually we discover

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 3 & 0 \\ 4 & 5 & 6 \end{bmatrix}. \tag{25}$$

However, we would never want to do this algorithm by hand. It is built in to most numerical mathematics software. For instance, in MATLAB the command is "chol." To use is, you type in

```
>> L = chol(A, 'lower')
```

to get the lower triangular matrix of the Cholesky factorization of matrix A. MATLAB does its best to make this computation in a numerically stable way.

Now, we apply this to do orthogonalization of vectors (equivalently, the Gram-Schmidt process). Start with n linearly independent vectors $u_1, u_2, \ldots u_n$ in real space $\mathbb{R}^m$. Organize them into n columns, to create the matrix

$$U = [u_1|u_2|\ldots|u_n]\,. \tag{26}$$

The matrix U^*U is square ($n \times n$), symmetric, and even positive. We apply Cholesky decomposition to find a lower triangular matrix L with

$$U^*U = LL^*. \tag{27}$$

The fact is, the matrix $V = U(L^{-1})^*$ has columns which what we would get with the Gram-Schmidt process. That is we can say:

Theorem 3 *The matrix*

$$V = U(L^{-1})^* = [v_1|v_2|\ldots|v_n] \tag{28}$$

has n orthonormal columns $v_1, v_2, \ldots, v_n$ which span the same subspace as the original vectors $u_1, u_2, \ldots, u_n$.

In fact, since the matrix L and its inverse L^{-1} are lower triangular, the transpose $(L^{-1})^*$ is upper triangular, so the theorem shows the first k columns of V are a linear combination of the first $k$columns of U, for each integer k. That is, $v_1 \ldots v_k$ span the same space as $u_1 \ldots u_k$, for each k, just like with Gram-Schmidt.

We can see the columns of V are orthogonal by compute the dot product of each column with all the others: equivalently, we just check that the matrix product V^*V is the identity matrix:

$$V^*V = (U(L^{-1})^*)^*U(L^{-1})^* = L^{-1}U^*U(L^{-1})^* = L^{-1}LL^*(L^{-1})^* = I \cdot I = I, \tag{29}$$

as desired.

3.2 Apply Choleski to functions

So, how do we apply this functions, to get an orthonormal set of functions? The key is to use the inner product to convert a "matrix of functions" into a matrix of numbers, apply Cholesky to that matrix, and then your good to go.

It's probably easiest to demonstrate this with an example. For our vector space, $C[-1,1]$, the set of continuous functions on the interval $[-1,1]$. The inner product comes from the usual integral on the same interval. Suppose we want to find a orthonormal basis (of functions) which span the same space as the four monomial functions $1, x, x^2, x^3$. Start with a matrix U whose four columns are these four functions,

$$U = \left[1|x|x^2|x^3\right]. \tag{30}$$

The product U^*U is obtained by taking inner products of the columns with themselves:

$$U^*U = \begin{bmatrix} \dfrac{1}{} \\ \hline x \\ \hline x^2 \\ \hline 0 \end{bmatrix} \left[1|x|x^2|x^3\right] = \begin{bmatrix} \langle 1,1\rangle & \langle 1,x\rangle & \langle 1,x^2\rangle & la1,x^3\rangle \\ \langle x,1\rangle & \langle x,x\rangle & \langle x,x^2\rangle & \langle x,x^3\rangle \\ \langle x^2,1\rangle & \langle x^2,x\rangle & \langle x^2,x^2\rangle & \langle x^2,x^3\rangle \\ \langle x^3,1\rangle & \langle x^3,x\rangle & \langle x^3,x^2\rangle & \langle x^3,x^3\rangle \end{bmatrix}. \tag{31}$$

These inner products are easy enough to compute, since we know that

$$\langle x^m, x^n\rangle = \int_{-1}^{1} x^{n+m}\,dx = 2/(m+n+1) \tag{32}$$

when $m+n$ is even, and equals zero otherwise.

This matrix of inner products is what we use for the Cholesky decomposition, so in MATLAB we find

```
 >> C = [2 0 2/3 0; 0 2/3 0 2/5; 2/3 0 2/5 0 ; 0 2/5 0 2/7]

C =

    2.0000         0    0.6667         0
         0    0.6667         0    0.4000
    0.6667         0    0.4000         0
         0    0.4000         0    0.2857

>> L = chol(C,'lower')

L =

    1.4142         0         0         0
         0    0.8165         0         0
    0.4714         0    0.4216         0
         0    0.4899         0    0.2138

>> inv(L)'

ans =

    0.7071         0   -0.7906         0
         0    1.2247         0   -2.8062
         0         0    2.3717         0
         0         0         0    4.6771
```

Now, we interpret the "function matrix " product

$$U(L^{-1})^* = \left[1|x|x^2|x^3\right] \begin{bmatrix} 0.7071 & 0 & -0.7906 & 0 \\ 0 & 1.2247 & 0 & -2.8062 \\ 0 & 0 & 2.3717 & 0 \\ 0 & 0 & 0 & 4.6771 \end{bmatrix} \tag{33}$$

as a linear combination of the monomials $1, x, x^2, x^3$. Thus our four orthonormal functions are

$$0.7071, 1.2247 \cdot x, -0.7906 + 2.3717 \cdot x^2, -2.8062 \cdot x + 4.6771 \cdot x^3. \tag{34}$$

That is, we have these four simple polynomials that are orthonormal in $C[-1, 1]$ with the usual inner product. (As an exercise, you can try computing their inner products, to ensure they are orthogonal.)

Up to a scaling factor, these are the Legendre polynomials. You might like to try finding the first 10 orthogonal polynomials generated by $1, x, x^2, \ldots, x^9$, and compare your answer with a table of Legendre polynomials. (Or look them up on Wikipedia.)

The Hermite polynomials are obtained in a similar fashion, by orthogonalizing the monomials $1, x, x^2, \ldots$ in the space of continuous functions on the real line, using the (weighted) inner product

$$\langle f, g \rangle = \int_{-\infty}^{\infty} f(x)g(x)e^{-x^2}\, dx. \tag{35}$$

As you might expect from the Gaussian appearing in the integral, Hermite polynomials are interesting both in statistics (probability) and in quantum physics. (Again, you can look this up in Wikipedia.) Again, as an exercise, you can try to compute the first few Hermite polynomials.

What's the point of all this: you can use matrix methods, and numerical packages, to do the dirty work for you in orthogonalizing functions.

We often are interested in orthogonal wavelets, so it is important to know how to make certain functions orthogonal.

3.3 Linear transformations, eigenvalues, eigenvectors

I think I said something about linear transformations, eigenvalues, and eigenvectors. Students will have seen this for matrices. But it is also useful for more general linear transformations. So, for instance, on the set of differentiable functions $C^\infty[-1, 1]$, we can use the derivative to define a linear transformation T on functions f as

$$Tf = \frac{df}{dx}. \tag{36}$$

Eigenvectors will be functions that satisfy

$$Tf = \lambda f, \text{ equivalently } \frac{df}{dx} = \lambda f. \tag{37}$$

The solution to this differential equation is the exponential with parameter λ, so

$$f(x) = a \cdot e^{\lambda x} \tag{38}$$

gives a whole family of eigenvectors (eigenfunctions). Weird that there are no matrices here, but we are still talking about eigen-thingies.

4 Lecture 4: Block matrices

When working with matrices with many entries, it can be convenient to partition them into blocks. So long as the block sizes are compatible, all the obvious operations work out just fine.

$$\left[\begin{array}{ccc|cc} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ \hline 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \end{array}\right] \left[\begin{array}{cc} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ \hline 7 & 8 \\ 9 & 10 \end{array}\right] = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E \\ F \end{bmatrix} = \begin{bmatrix} AE + BF \\ CE + DF \end{bmatrix} \tag{39}$$

This can be convenient for certain algorithms.

It is useful to note that for block diagonal matrices, many operations simplify in the obvious way. A block diagonal matrix looks like this:

$$\begin{bmatrix} A & 0 & 0 \\ 0 & B & 0 \\ 0 & 0 & C \end{bmatrix} \tag{40}$$

where the sub matrices along the diagonal (A,B,C) are square, possibly of different dimensions. We have several nice properties, including

- the inverse of the block diagonal matrix is the matrix obtained by taking the inverse of each sub matrix block along the diagonal;
- the determinant of the block matrix is the product of the determinants of each block;
- the eigenvalues of the block matrix is the union of the eigenvalues of each block;
- the corresponding eigenvectors are just the eigenvectors of each block, extended to the proper dimension by appending zeros in the obvious places;
- a block matrix is orthogonal (or unitary) if and only if each block is orthogonal (or unitary).

In fact, similar results are true for upper triangular block matrices, like this:

$$\begin{bmatrix} A & B & C \\ 0 & D & E \\ 0 & 0 & F \end{bmatrix}. \tag{41}$$

In this case, the determinant is the product of the determinants of each block, and the eigenvalues are the union of the eigenvalues of the block. However, the inverses and eigenvectors are all messed up!

It is an interesting exercise to try to compute the inverse of an upper triangular block matrix. One might guess that

$$\begin{bmatrix} A & B \\ 0 & C \end{bmatrix}^{-1} = \begin{bmatrix} A^{-1} & D \\ 0 & C^{-1} \end{bmatrix}, \tag{42}$$

for some matrix D. Multiplying the first matrix with the second convinces us that

$$AD + BC^{-1} = 0 \tag{43}$$

from which we conclude $D = -A^{-1}BC^{-1}$. Thus

$$\begin{bmatrix} A & B \\ 0 & C \end{bmatrix}^{-1} = \begin{bmatrix} A^{-1} & -A^{-1}BC^{-1} \\ 0 & C^{-1} \end{bmatrix}. \tag{44}$$

As an exercise, you might try to compute the inverse of an upper triangular matrix with 3 block sub-matrices on the diagonal.

Wavelet transformations usually give matrices that have big blocks of zeros. For instance, the Haar wavelet transform is obtained by sampling some square waves (boxcar functions) that take values one, zero, or minus one. I guess I should draw a picture here, but it's late and pictures are a pain in Latex. So, here are the four vectors I have in mind:

$$(1,1,1,1), (1,1,-1,-1), (1,-1,0,0), (0,0,1,-1). \tag{45}$$

Notice these are mutually orthogonal. Organize them into column vectors of length one, and we get the Haar wavelet transform matrix

$$W = \begin{bmatrix} .5 & .5 & .707 & 0 \\ .5 & .5 & -.707 & 0 \\ .5 & -.5 & 0 & .707 \\ .5 & -.5 & 0 & -.707 \end{bmatrix}. \tag{46}$$

For any dimension n that is a power of 2, you can write down a n by n matrix that is obtained in this way.

In class, Eric asked why powers of 2. Well, it is easy to construct as powers of 2, but there are also other dimensions that are possible. There is a more interesting general answer. Hadamard matrices are square matrices with entries ± 1 (and no zeros), whose columns happen to be orthogonal. Raymond Paley conjectured that there exist such Hadamard matrices of dimension 4k by 4k, for any integer k. The smallest dimension for which such a matrix has not yet been found is 668. Find it and you'll be (moderately) famous.

Paley is a dead mathematician from Cambridge, buried in the town cemetery in Banff. He died in 1933 in an avalanche on Fossil mountain, just outside the Banff townsite. You can go visit his grave, if you like that kind of thing! It's easy enough to construct Hadamard matrices of size $2^n \times 2^n$: at dimension 1, take $H_1 = [1]$. Now work your way up by factors of two, by constructing each higher dimension by the iteration

$$H_{n+1} = \begin{bmatrix} H_n & H_n \\ H_n & -H_n \end{bmatrix}. \tag{47}$$

You should verify this bigger block matrix H_{n+1} has orthogonal columns, provided the blocks H_n do as well.

Hadamard matrices come up in all sorts of places, but sadly I don't think we need them for wavelets.

5 Lecture 5: Signals in 1D and 2D. Sounds and images

5.1 Signals and systems

In signal processing, we are mainly interested in *signals* and *systems*. A typical signal might be a sound, a bit of music, a photographic image, a seismic vibration, or just about any physical phenomena that can be measured over time and/or space. Usually, we represent a signal by a function $f(t)$, where t is time, and $f(t)$ is the value measured at time t. For an image, we would have a function of two variables $f(x, y)$, where f represents the intensity of the image at position x, y in the plane, say.

A *system* takes one signal in, and outputs another signal. It could be a physical system: an earthquake in India starts a signal (vibrations) on one side of the earth, the earth transmits the vibrations to the other side (the system), and a new signal is felt in Calgary (the received vibrations). It could be an electrical system: a sound is picked up by a microphone (the input signal), the signal is passed to a stereo amplifier (the system), and the resulting amplified signal is output to the speakers (the output signal). It could be a computational system: a string of numbers is input to a computer, the computer churns away on the numbers (adding, subtracting, multiplying, etc – the system), and a string of numbers is output by the computer. It could be a combination of such systems: a digital camera captures a real image through its lens, the intensities are converted to a function $f(x, y)$, and then the computer mucks around with the values of the function to compute a sharper image, represented by a new function $g(x, y)$. Here, the input is the real image, the system is the camera/computer, the output is the function $g(x, y)$.

The point is: signals are functions, and systems operate on signals. Our wavelet transforms will act on signals.

5.2 Sampling

The signal $f(t)$ is a function on the real line $\mathbb{R}$. When we compute with a computer, usually we can't know everything about the function, or store it all on the computer. So we just evaluate the signal at a sequence of times t_n, and define a vector $\mathbf{x} = (\ldots, x_{-2}, x_{-1}, x_0, x_1, x_2, \ldots)$ with components

$$x_n = f(t_n), \qquad \text{for all } n \in \mathbb{Z}. \tag{48}$$

For reasons that have mostly to do with engineering technology, we usually take the time samples t_n to be uniformly spaced. That is, we have a sequence of numbers separated by a uniform step Δt, and write $t_0 = 0, t_1 = \Delta t, t_2 = 2\Delta t, \ldots$, and so the vector $\mathbf{x}$ has components

$$x_n = f(n\Delta t), \qquad \text{for all } n \in \mathbb{Z}. \tag{49}$$

Although the vector $\mathbf{x}$ is infinitely long, that is it has infinitely many components, it is important to realize that most of the component x_n are just zero. Why? Because we can't measure back to time minus infinity, or forward to plus infinity. So at some point we stop measuring, and can just assume everything else is zero.

The vector $\mathbf{x}$ is called a sampled signal. Δt is called the sampling interval. $1/\Delta t$ is called the sampling rate.

5.3 Quantization

We like to work with real numbers, which have infinite accuracy, but computers work with fixed point or floating point numbers, that have only finite accuracy. The sampled data $f(n\Delta t)$ is rounded to the nearest fixed point number that our equipment can handle, and stored in the computer.

Over the years, the engineers have made better and better equipment, so quantization is getting better and better. Back in the day (e.g. 1970's), I used audio digitizers that stored numbers as integers that range between -128 and +127. (These are the numbers that can be stored in 8 bits, or a single byte in the computer.) The roundoff error in quantizing is about 1%. On a modern compact disk (CD), data is stored as 16 bits, giving a range of -32,768 to 32767. The roundoff error is much smaller. On high quality audio equipment, it is common to use 24 bit equipment nowadays; even 32 nbit is possible. Much more accurate, but still only finite accuracy.

5.4 Aliasing

The problem with sampling and quantizing is that you lose information in the process. Two different functions $f(t)$ and $g(t)$ might get sampled and produce the same vector $\mathbf{x}$. For instance, suppose f is a sine wave, $f(t) = \sin(t)$ and g is the zero function, $g(t) = 0$. These are two very different signals. But, with the sampling interval $\Delta t = \pi$, we see that

$$\begin{aligned} x_n &= f(n\pi) = \sin(n\pi) \\ &= 0 \\ &= g(n\pi), \end{aligned}$$

so f and g get sampled to appear as the same vector $\mathbf{x}$, which happens to be the zero vector. This is called *sampling aliasing*: one signal appears the same as another.

Another form of error comes from the quantization process. Two different functions $f(t)$ and $g(t)$ might differ by an amount so small that in the round-off process, the samples become the same. In general, the quantized samples x_n of the function samples $f(n\Delta t)$ can be expressed as sum of the function, plus a small error:

$$x_n = f(n\Delta t) + \epsilon_n. \tag{50}$$

Typically, we treat this unknown ϵ_n as a small amount of noise added into the system.

5.5 Frequency aliasing

There is a special kind of aliasing, where two sinusoidal signals appear the same under sampling. A complex sinusoid is a function of the form

$$f(t) = e^{2\pi i\omega t} = \cos(2\pi\omega t) + i\sin(2\pi\omega t). \tag{51}$$

This signal is periodic, which repeats itself at a rate of ω cycles per unit time. Eg. $f(t) = e^{2\pi i 60t}$ represents a 60 Hertz signal, where t is measured in seconds. Two signals

$$f(t) = e^{2\pi i\omega_1 t} \qquad g(t) = e^{2\pi i\omega_2 t}, \tag{52}$$

will get aliased at a sample interval Δt if

$$f(n\Delta t) = g(n\Delta t) \text{ for all integers } n. \tag{53}$$

Equivalently,

$$(e^{2\pi i\omega_1\Delta t})^n = (e^{2\pi i\omega_2\Delta t})^n \text{ for all integers } n, \tag{54}$$

or more simply, if $e^{2\pi i\omega_1\Delta t} = e^{2\pi i\omega_2\Delta t}$. A bit of algebra shows this happens if $\omega_1 - \omega_2 = N/\Delta t$, for some integer N.

Thus, two sinusoids get aliased if the difference of their frequencies $\omega_1 - \omega_2$ is a multiple of the sampling rate $1/\Delta t$.

Usually, we are interested in measuring signals with frequencies in some interval $[-F, F]$. So, for instance, you might like to measure frequencies ω_1, ω_2 in $[-400Hz, 400Hz]$. The difference $\omega_1 - \omega_2$ could be as big as 800Hz. In order for this to not be a multiple of the sample rate, we have to choose a sample rate bigger than 800Hz. That is,

$$800Hz \le \frac{1}{\Delta t}. \tag{55}$$

Being a little big lazy, we might choose the sample rate to be 1000Hz, and so the sampling interval is $\Delta t = .001$ second (a millisecond).

5.6 Sampled signals as a vector space

The sampled signal

$$\mathbf{x} = (\ldots, x_{-2}, x_{-1}, x_0, x_1, x_2, \ldots) \tag{56}$$

is supposed to look like a vector, just as you learned in linear algebra. It just happens to be infinitely long. You can still treat as you would regular vectors: add, subtract, pointwise multiply, scalar multiply, and take inner products. For instance, write

$$\begin{aligned}
\mathbf{x} &= (\ldots, 0, 0, 1, 2, 3, 0, 0, \ldots) \\
\mathbf{y} &= (\ldots, 0, 0, 4, 5, 6, 0, 0, \ldots) \\
\mathbf{x}+\mathbf{y} &= (\ldots, 0, 0, 5, 7, 9, 0, 0, \ldots) \\
\mathbf{y}-\mathbf{x} &= (\ldots, 0, 0, 3, 3, 3, 0, 0, \ldots) \\
\mathbf{x}\cdot\mathbf{y} &= (\ldots, 0, 0, 4, 10, 18, 0, 0, \ldots) \\
10\mathbf{x} &= (\ldots, 0, 0, 10, 20, 30, 0, 0, \ldots) \\
10\mathbf{y} &= (\ldots, 0, 0, 40, 50, 60, 0, 0, \ldots) \\
\langle \mathbf{x}, \mathbf{y} \rangle &= 1\cdot 4 + 2\cdot 5 + 3\cdot 6 = 32.
\end{aligned}$$

It is a nuisance to keep writing all these zeros in the infinite vectors, so sometimes we shorten things to condensed vectors, like $\mathbf{x} = (1, 2, 3)$ and $\mathbf{y} = (4, 5, 6)$. In this form, we have the 0-th entry in $\mathbf{x}$ as the first number that appears in the short vector. So $x_0 = 1, x_1 = 2, x_2 = 3$, and the rest are zero.

5.7 Photos as 2D signals

Digital photos are stored in the computer as an array of numbers. For instance, the simple 7x5 matrix of this form

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{57}$$

can be displayed as a simple, chunky happy face. A bigger matrix makes a more detailed picture, as shown by the photo of a sketch by Durer in Figure 1.

In general, we think of a black-and-white image as being represented by a function of 2 variables, $f(x, y)$. As with 1D signals, we sample on a grid, but with spacings in both the x and y directions,

$$X_{m,n} = f(m\Delta x, n\Delta y). \tag{58}$$

This sampling will lead to sampling error, and finer resolutions are obtained by shrinking the Δx and Δy. The values in the array $X_{m,n}$ are quantized, which can lead to quantization error. On a modern computer, typically the values stored are integers in the range of 0 to 255. (8 bits resolution per pixel).

Color images are stored as 3 arrays of values, represent intensities in the three colour channels red, green, and blue:

$$R_{m,n} = r(m\Delta x, n\Delta y), G_{m,n} = g(m\Delta x, n\Delta y), B_{m,n} = b(m\Delta x, n\Delta y). \tag{59}$$

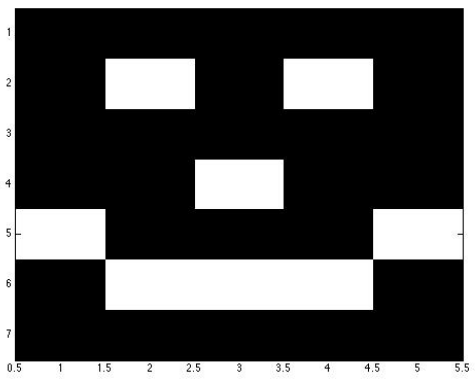
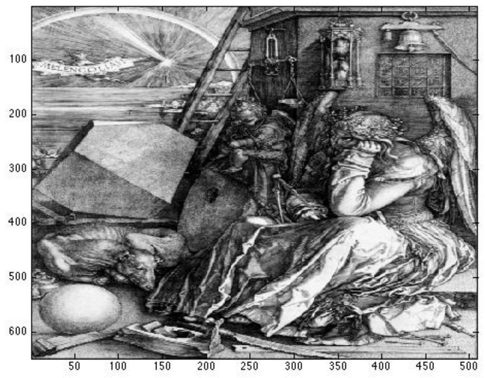



Figure 1: A 7x5 happy face, and a 600x500 detailed photo.

This choice of colour channels has to do with the way the human eye works – we have three types of colour receptors on our retina that are sensitive to these three particular colors. Our digital image only has to keep track of these three channels.

For example, we can decompose this nice flu jet colour image into 3 components, as in Figure 2. Note where there is high intensity red, for instance, we see a corresponding white patch on red component, and similar with the other components.

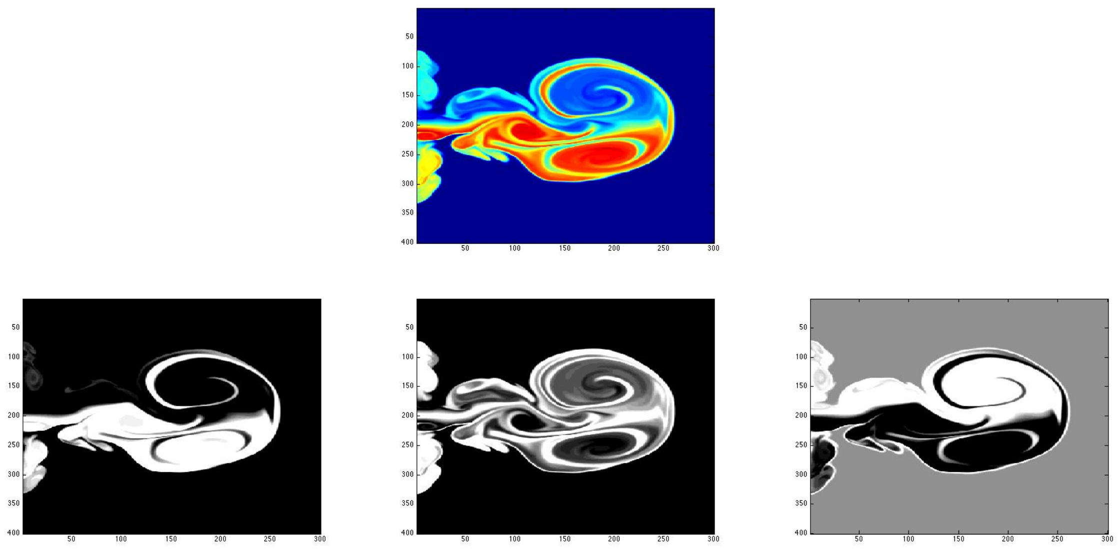


Figure 2: A full colour image, decomposed into red, green, and blue components.

Note that MATLAB often stores images in a compressed format, with a colour map and a set of indices that refers to particular colours. When doing image processing in MATLAB, it is important to convert the compressed format into proper intensity matrices of the there RGB channels.

There are also ways to store images in HSI format (Hue, Saturation, Intensity), as well as YCbCr format (Y=luminance, Cb, Cr - colour difference channels). The YCbCr format is particularly useful in data compression.

5.8 Simple operations on digital photos

You can make simple transformations on a photo by operating pixel-by-pixel.

For instance, to get a "negative" of an image, just reverse all values from 0 through 255:

$$Y = 255 - X. \tag{60}$$

This form assumes the X values all lie between 0 and 255.

To do a gamma correction, set

```
 Y = 255*(X/255).^g,
```

where $g = \gamma$ is some positive value. Smaller g values, less than one, tend to lighten an image. Larger g values, greater than one, tend to darken an image.

Two dimensional convolution gives a simple filter of an image. For instance, if we convolve an image with a small 3x3 matrix of ones, gives a blurred version of the original image. In MATLAB,

$$Y = conv2(X, [1,1,1;1,1,;1,1,1]). \tag{61}$$

Again, try it out.

5.9 Energy calculations

Since we will be interested in compressing signals and images, we will carefully define what we mean by energy of a signal. Essentially, it is a normalized length squared.

For a 1D signal of length N, $Energy(X) = \frac{1}{N}\sum |x_n|^2$.

For a 2D signal of length $M \times N$, $Energy(X) = \frac{1}{MN}\sum |x_{m,n}|^2$.

When a measured signal consists of two parts, real signal and noise, the ratio is called the Signal-to-Noise ration (SNR). This is usually measure in decibels (dB), by the definition

$$SNR = 10\log_{10}\frac{Energy(signal)}{Energy(noise)} \text{ decibels.} \tag{62}$$

It's interesting to observe the ratio of signal to quantization error is about

$$SNR = 10\log_{10} 2^{2Q} \approx 6.02 \cdot Q \quad dB, \tag{63}$$

where Q is the number of bits in the quantizer. SO, an 8-bit quantizer has an SNR of 48dB, while a 16-bit quantizer has an SNR of 96dB. A 24-bit quantizer has a theoretical SNR of 144.5dB, but this is pretty hard to achieve with current hardware.

The book makes the point of defining the peak signal-to-noise ratio for an image X and a an approximate reconstruction Y, with 0 to 255 integer encoding, as

$$PSNR = 10\log_{10}\frac{255^2}{Energy(X-Y)} \text{ decibels.} \tag{64}$$

In my opinion, this is a silly definition: it would be better to take the usual SNR, using the difference $X - Y$ as the error. That is, we write

$$SNR = 10\log_{10}\frac{Energy(X)}{Energy(X-Y)} \text{ decibels.} \tag{65}$$

Try them both out, see which seems to be more informative to you.

Finally, we define the cumulative energy function. This is just a vector that shows how quickly the larger components of a signal fill up the total energy of the signal. Given a signal X, 1D or 2D, order the components from big to small (in absolute value) and write it as a vector $(y_1, y_2, y_3, \ldots, y_n)$. The cumulative energy is a vector $C = (c_1, c_2, c_3, \ldots, c_n)$ with each component defined as

$$c_k = \frac{|y_1|^2 + |y_2|^2 + \cdots + |y_k|^2}{|y_1|^2 + |y_2|^2 + \cdots + |y_k|^2 + \cdots + |y_n|^2}. \tag{66}$$

The cumulative energy starts near zero and runs up to one. The faster it gets near one, the more likely it is we can compress by throwing away the small components.

6 Lecture 6: The Fourier transform.

The Fourier transform is simply a method of converting a function on an interval, into a sequence of coefficients, and vice-versa.

The Fourier transform of a function $f(\omega)$ on the interval $[-1/2, 1/2]$ is defined as a sequence $\mathbf{x} = (\ldots, x_{-1}, x_0, x_1, x_2, \ldots)$ whose coefficients x_n are given by the integral

$$x_n = \int_{-1/2}^{1/2} f(\omega)e^{-2\pi i n\omega}\, d\omega. \tag{67}$$

You might recognize this as the inner product $x_n = \langle f(\cdot), e^{2\pi i n(\cdot)}\rangle$ on the space of function on the interval $[-1/2, 1/2]$. Here, we are using the sequence of exponential functions

$$e^{2\pi i n\omega} = \cos(2\pi n\omega) + i \cdot \sin(2\pi n\omega) \tag{68}$$

which turns out to be an orthonormal set of functions in the space $C([-1/2, 1/2])$, which you can easily check.

Conversely, given a sequence $\mathbf{x} = (\ldots, x_{-1}, x_0, x_1, x_2, \ldots)$, we define a function $f(\omega)$ on the interval $[-1/2, 1/2]$ by the infinite sum

$$f(\omega) = \sum_{n=-\infty}^{\infty} x_n e^{2\pi i n\omega}. \tag{69}$$

The x_n are called the Fourier coefficients of function f. Conversely, the sum $\sum x_n e^{2\pi i n\omega}$ is called the Fourier series. Remarkably, these are inverse operations: starting with a function f, you compute the Fourier coefficients, and from the coefficients, you compute the Fourier series. The series converges (almost everywhere) to the original function f.

This is not too surprising for functions that are finite linear combinations of the exponentials – since they form an orthonormal set, and this is just an expansion in that basis. With some technical assumptions, we can show the expansion works for a wide class of functions. (Certainly for bounded, piecewise continuous functions.)

We will use such sequences to construct our wavelet filter banks.

As an example, the sequence

$$x_{-1} = 3, x_0 = 5, x_1 = 7, x_2 = 9 \tag{70}$$

has Fourier series

$$f(\omega) = 3e^{-2\pi i\omega} + 5 + 7e^{2\pi i\omega} + 9e^{4\pi i\omega}, \qquad -1/2 \le \omega \le 1/2. \tag{71}$$

Try to verify that the resulting Fourier coefficients

$$\int_{-1/2}^{1/2} f(\omega) e^{-2\pi i n\omega}\, d\omega. \tag{72}$$

are exactly the x_n we began with.

6.1 The Fast Fourier transform

The Fast Fourier Transform (FFT) just evaluates the FT of a finite length sequence $(x_0, x_1, \ldots, x_{N-1})$ on fractional values of normalized frequency $\omega = 0, 1/N, 2/N, \ldots (N-1)/N$ in the form

$$X(\frac{k}{N}) = \sum_{j=0}^{N-1} x_n e^{2\pi i jk/N}, \qquad 0 \le k \le N-1. \tag{73}$$

Note the limits of the sum: the FFT applies to sequences of length N, starting at term x_0. The inversion formula reduces to a sum,

$$x_j = \frac{1}{N} \sum_{k=0}^{N-1} X(\frac{k}{N}) e^{-2\pi i jk/N}, \qquad 0 \le j \le N-1. \tag{74}$$

Again, this only works for vectors of length N. Note we have N input values, and N output values, and this is a linear, invertible transformation. Up to proper normalization, it is actually a unitary transformation.

It is interesting to write down the matrix of this transformation. It is simply a symmetric matrix (it equals its own transpose) whose entries are complex exponentials, spaced uniformly around the unit circle. The matrix entries are

$$M_{jk} = \exp(2\pi i \frac{jk}{N}), \qquad 0 \leq j,k \leq N-1. \tag{75}$$

The inverse matrix is the complex conjugate of this matrix, divided by the factor N.

You should write out this matrix for $N = 4$ and $N = 8$ to see that you get the main idea. Check that the columns are orthogonal (in the complex inner product).

When computing a Fourier transform, you should never compute nor store this matrix in the computer – that is too wasteful. There is a fast algorithm, called the fast Fourier transform, that quickly computes the transform for you, and avoids the N by N matrix storage.

It's called the FAST Fourier transform because this algorithm that reduces the computation from about N^2 operations, to only $N \log_2 N$ operations. For a short seismic trace of $N = 1000$ samples, this means instead of one million operations on the computer, we only need about 10,000 – a speed-up of a factor of 100.

When I was your age, I had to write a FFT code in assembly code for the Motorola 68020 CPU in the early Macintosh computer – no floating point or code libraries for us code jockeys back then! I used a version of the FFT algorithm called the Hartley transform, that doesn't use complex numbers. Nowadays, you should just use optimized code that someone else writes – like the code in MATLAB. Or, use the code in fftw.com, the fastest FFT code in the west. (Seriously!)

Keep in mind, the true Fourier transform of a sequence $(x_0, x_1, \ldots, x_{N-1})$ is a sum of exponentials – so it is a continuous, periodic function on the unit interval. If you want plot this continuous function, taking the FFT will only give a few sample points of the curve – to get a smooth plot of the function, you want MANY sample points. Fast way to do this is pad the sequence $(x_0, x_1, \ldots, x_{N-1})$ with a bunch of zeros, then plot the resulting FFT. All you've done is made many more sample points on the curve.

In MATLAB,

```
x = [ 1 2 3 ];
y = [x, zeros(1,100)];
plot(real(fft(y)))
```

to plot just over 100 points of the real part of the curve

$$f(\omega) = 1 + 2e^{2\pi i\omega} + 3e^{4\pi i\omega}. \tag{76}$$

6.2 MATLAB's FFT

MATLAB indexes its vectors from 1 to N, and so the FFT is expressed with shifted indices. With a vector $\mathbf{x} = [x(1), x(2), \ldots, x(N)]$ and FFT $X = [X(1), X(2), \ldots, X(N)]$, MATLAB will compute

$$X(k) = \sum_{n=1}^{N} x(n) e^{2\pi i (n-1)(k-1)/N}, \qquad 1 \le k \le N. \tag{77}$$

The inverse ifft is computed as

$$x(n) = \frac{1}{N} \sum_{k=1}^{N} X(k) e^{-2\pi i (n-1)(k-1)/N}, \qquad 1 \le n \le N. \tag{78}$$

6.3 Symmetries of the Fourier transform

When the inputs sequence $\mathbf{x}$ is real (i.e. we use real data, not complex), the Fourier series $f(\omega)$ is complex valued, but have the amazing symmetry:

$$f(-\omega) = \overline{f(\omega)}. \tag{79}$$

That is, the values at negative frequencies are the complex conjugate of the values at positive frequencies. (WHY? Check yourself!) It also means the absolute value of the values at negative frequencies are the same as those at the corresponding positive frequency:

$$|f(-\omega)| = |f(\omega)|. \tag{80}$$

Because of periodicity, we can also say

$$|f(1-\omega)| = |f(\omega)|, \text{ for } 0 \le \omega \le 1. \tag{81}$$

You will always see this symmetry in your MATLAB plots of Fourier transforms. Keep in mind the symmetries in MATLAB will be slightly difference, since we start "counting" our indices at $n = 1$, corresponding to frequency $\omega = 0$, and ending at $n = N$ corresponding to frequency $\omega = 1 - 1/N$.

7 Lecture 7: Convolution

Any two signals $\mathbf{x}, \mathbf{y}$ can be combined in a special operation called *convolution*. An easy way to define the convolution is using the Z transform. So, for example, if

$$\begin{aligned} \mathbf{x} &= (\ldots, 0, 0, 1, 2, 3, 0, 0, \ldots) \\ \mathbf{y} &= (\ldots, 0, 0, 4, 5, 6, 0, 0, \ldots), \end{aligned}$$

we have Z transforms $X(Z) = 1 + 2Z + 3Z^2, Y(Z) = 4 + 5Z + 6Z^2$. Take the product of the two polynomials,

$$(1 + 2Z + 3Z^2)(4 + 5Z + 6Z^2) = 4 + 13Z + 28Z^2 + 27Z^3 + 18Z^4, \tag{82}$$

which we can recognize as the Z transform of the vector

$$(\ldots, 0, 0, 4, 13, 28, 27, 18, 0, 0, \ldots) \tag{83}$$

which we define as the convolution of $\mathbf{x}$ with $\mathbf{y}$. That is,

$$\mathbf{x} * \mathbf{y} = (\ldots, 0, 0, 4, 13, 28, 27, 18, 0, 0, \ldots). \tag{84}$$

There is a general formula for convolution: if you think about how polynomial multiplication works, it is pretty easy to see that the n-th entry in the vector $\mathbf{x} * \mathbf{y}$ will be a sum of terms like $x_j y_k$, where $j + k = n$. In other words, we can write

$$(\mathbf{x} * \mathbf{y})_n = \sum_k x_{n-k} y_k. \tag{85}$$

Because we know how polynomial multiplication works, we can observe that convolution works in either order and gives the same answer: $\mathbf{x}{*}\mathbf{y} = \mathbf{y}{*}\mathbf{z}$. Also, the convolution operation distributes over addition: $\mathbf{x}{*}(\mathbf{y}{+}\mathbf{z}) = (\mathbf{x}{*}\mathbf{y}){+}(\mathbf{x}{*}\mathbf{z})$: because multiplication of polynomials distributes over addition. You can also move in scalar constants quite freely, so for instance, $3(\mathbf{x} * \mathbf{y}) = (3\mathbf{x}) * \mathbf{y} = \mathbf{x} * (3\mathbf{y})$, which is clear from the summation formula defining convolution.

7.1 Convolution as matrix-vector multiplication

Notice we can organize a matrix and a vector to get the same result as the convolution in the last section. We just do the simple $\mathbf{x}, \mathbf{y}$ example:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 & 0 \\ 3 & 2 & 1 & 0 & 0 \\ 0 & 3 & 2 & 1 & 0 \\ 0 & 0 & 3 & 2 & 1 \end{bmatrix} \begin{bmatrix} 4 \\ 5 \\ 3 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 4 \\ 13 \\ 28 \\ 27 \\ 18 \end{bmatrix}. \tag{86}$$

The matrix representing $\mathbf{x}$ is constant along diagonals. That is called a Toeplitz matrix. $\mathbf{y}$ is organized as a column vector. The usual matrix-vector product that we learned in linear algebra gives the resulting column vector representing $\mathbf{x} * \mathbf{y}$.

Notice that although $\mathbf{x}, \mathbf{y}$ had only three non-zero entries, the matrix had to be 5 by 5, and the vector 5 by 1, in order for the arithmetic to work out. Similarly, for vectors with N non-zero entries (all in a row), we need matrices of size 2N + 1.

7.2 Convolution as matrix-matrix multiplication

We observe that we can also represent convolution as the product to two Toeplitz matrices:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 & 0 \\ 3 & 2 & 1 & 0 & 0 \\ 0 & 3 & 2 & 1 & 0 \\ 0 & 0 & 3 & 2 & 1 \end{bmatrix} \begin{bmatrix} 4 & 0 & 0 & 0 & 0 \\ 5 & 4 & 0 & 0 & 0 \\ 6 & 5 & 4 & 0 & 0 \\ 0 & 6 & 5 & 4 & 0 \\ 0 & 0 & 6 & 5 & 4 \end{bmatrix} = \begin{bmatrix} 4 & 0 & 0 & 0 & 0 \\ 13 & 4 & 0 & 0 & 0 \\ 28 & 13 & 4 & 0 & 0 \\ 27 & 28 & 13 & 4 & 0 \\ 18 & 27 & 28 & 13 & 4 \end{bmatrix}. \tag{87}$$

Check the matrix multiplication; you will see this is correct. Again, the matrices have to be large enough to give room for the whole convolution to appear.

You could write out infinite matrices, but it's too hard to typeset at the moment!

7.3 Convolution by flipping and shifting

You can also obtain a convolution by flipping the order of one of the vectors, taking point-wise products, and sum. So, for instance, with the same $\mathbf{x}, \mathbf{y}$ as in the previous section, we flip around the $\mathbf{y}$ and write it underneath the $\mathbf{x}$ vector:

$$\begin{matrix} \ldots 0 & 0 & 1 & 2 & 3 & 0 \ldots \\ \ldots 6 & 5 & 4 & 0 & 0 & 0 \ldots \end{matrix}. \tag{88}$$

The pointwise product is

$$\ldots 0 \quad 0 \quad 4 \quad 0 \quad 0 \quad 0 \ldots \tag{89}$$

which sums up to 4, the first component of the convolution. To get the second component, we shift $\mathbf{y}$ once, and line up the vectors as

$$\begin{matrix} \ldots 0 & 0 & 1 & 2 & 3 & 0 \ldots \\ \ldots 0 & 6 & 5 & 4 & 0 & 0 \ldots \end{matrix}. \tag{90}$$

The pointwise product is

$$\ldots 0 \quad 0 \quad 5 \quad 8 \quad 0 \quad 0 \ldots \tag{91}$$

which sums up to 13, the second component of the convolution.

To get the third component, we shift $\mathbf{y}$ again, and line up the vectors as

$$\begin{matrix} \ldots 0 & 0 & 1 & 2 & 3 & 0 \ldots \\ \ldots 0 & 0 & 6 & 5 & 4 & 0 \ldots \end{matrix}. \tag{92}$$

The pointwise product is

$$\ldots 0 \quad 0 \quad 6 \quad 10 \quad 12 \quad 0 \ldots \tag{93}$$

which sums up to 28, the third component of the convolution. And so on. This works in general.

7.4 Convolution as a system

Fixing a vector $\mathbf{h}$, we define a system that acts on signals $\mathbf{x}$ as

$$\mathbf{x} \to A(\mathbf{x}) = \mathbf{h} * \mathbf{x} = \mathbf{y}. \tag{94}$$

That is, for input signal $\mathbf{x}$, our system outputs a signal $\mathbf{y}$ that is computed as $\mathbf{y} = \mathbf{h} * \mathbf{x}$.

From our description of convolution as matrices, just like in linear algebra, we certainly expect that this system is linear. That is,

$$\begin{aligned} A(\mathbf{x}_1 + \mathbf{x}_2) &= A(\mathbf{x}_1) + A(\mathbf{x}_2), \\ A(\alpha \mathbf{x}) &= \alpha A(\mathbf{x}). \end{aligned}$$

This is easy to verify from the formulas for convolution. Equivalently, the first equation follows since convolution distributes over addition, $\mathbf{h} * (\mathbf{x}_1 + \mathbf{x}_2) = \mathbf{h} * \mathbf{x}_1 + \mathbf{h} * \mathbf{x}_2$, and the second equation follows since multiplication by a scalar commutes with polynomial multiplication.

7.5 The convolution theorem

Remarkably, the Fourier transform changes a convolution into a product. That is, if $\mathbf{x}.\mathbf{y}$ are sequence, with $\mathbf{z} = \mathbf{x} * \mathbf{y}$, then by taking Fourier transforms,we get

$$Z(\omega) = X(\omega) \cdot Y(\omega), \qquad -1/2 < \omega < 1/2, \tag{95}$$

where we are using X, Y, Z to denote the Fourier series for the corresponding sequences.

This is easy enough to check for a few simple examples: take $\mathbf{x}$ as the sequence with a 1 in exactly the $n = 3$ slot, zero everywhere else, and $\mathbf{y}$ with a 1 in the $n = 7$ slot. Their convolution $\mathbf{z}$ has a 1 exactly at the $n = 10$ slot. The convolution theorem asserts that

$$e^{2\pi i 10\omega} = e^{2\pi i 3\omega} e^{2\pi i 7\omega}, \tag{96}$$

which is obviously true since the arguments of the exponents just add.

The general case is shown by linearity.

We will use this result to design our wavelet filters with certain properties.

7.6 Linear, shift-invariant systems

We want to find out what are the linear, shift-invariant systems. Linear means the system is relatively simple in that sums of input signals map to sums of output signals, while shift-invariant means if we delay the input signal, the resulting output signal is the same as before, only delayed. This is also a reasonable assumption for a system that does not change over time.

We express these conditions on a system A in the following equations:

$$\begin{aligned} A(\mathbf{x}_1 + \mathbf{x}_2) &= A(\mathbf{x}_1) + A(\mathbf{x}_2), \text{ for all signals } \mathbf{x}_1, \mathbf{x}_2, \\ A(\alpha\mathbf{x}) &= \alpha A(\mathbf{x}), \text{ for all signals } \mathbf{x}, \text{ scalars } \alpha, \\ A(S\mathbf{x}) &= SA(\mathbf{x}), \text{ for all signals } \mathbf{x}, \end{aligned}$$

where S is the shift operator.

Theorem 4 *Suppose A is a linear, shift-invariant system. Then there is a vector $\mathbf{h}$ so that A is just convolution by $\mathbf{h}$. That is,*

$$A(\mathbf{x}) = \mathbf{h} * \mathbf{x}, \text{ for all signals } \mathbf{x}. \tag{97}$$

The proof will go like this. $\mathbf{h}$ is just A applied to the delta vector δ^0, the vector with zeros everywhere except at the $n = 0$ spot. We then use shift invariance to find that A acting on any δ^n is just that delta function, convolved with $\mathbf{h}$. We then use linearity to conclude A acts on any vector by convolving with $\mathbf{h}$.

The details are like this. Let $\mathbf{h} = A(\delta^0)$, which is a vector, since A acts on the given input vector δ^0 to produce some output vector, which we call $\mathbf{h}$. We note the shift operator S takes the vector δ^0 to δ^1, so by shift-invariance, we see

$$A(\delta^1) = A(S\delta^0) = S(A(\delta^0)) = S\mathbf{h} = \mathbf{h} * \delta^1, \tag{98}$$

since by definition, $A(\delta^0)$ is $\mathbf{h}$, and applying S to $\mathbf{h}$ is the same as convolving by δ^1.

Repeating this argument, we see that $A(\delta^n) = A(S^n\delta^0) = S^n A(\delta_0) = S^n\mathbf{h} = \mathbf{h} * \delta^n$. So now we know that A, applied to any of the delta vectors, just gives $\mathbf{h}$ convolved with the vector.

Now, any vector $\mathbf{x}$ is a linear combination of the δ^n, so by linearity of A, we have

$$A(\mathbf{x}) = A(\sum x_n\delta^n) = \sum x_n A(\delta^n) = \sum x_n \mathbf{h} * \delta^n = \mathbf{h} * (\sum x_n \delta^n) = \mathbf{h} * \mathbf{x}, \tag{99}$$

where at the second last equality, we use the fact that convolution by $\mathbf{h}$ is linear.

And that's it. If you want to worry about mathematical details, you should worry about whether these infinite sums converge. For our purposes, we can just assume the $\mathbf{x}$ is always given by a finite sum. (All but finitely many of the x_n are zero.)

7.7 Impulse response of a LSI system, Z transform

The vector $\mathbf{h}$ that appeared in the last section is call the *impulse response* of the system A. From an engineering point of view, it is the response of the system to getting a whack at time $t = 0$. Knowing the impulse response basically tells you everything you need to know about the linear, shift invariant system.

The Z transform of the system A is given as the Z transform of the impulse response **h**, which is the polynomial

$$H(Z) = \sum_k h_k Z^k. \tag{100}$$

7.8 FIR systems, minimum phase, maximum phase

A Finite Impulse Response (FIR) system is a linear, shift-invariant system where the impulse response function **h** only has finitely many non-zero coefficients h_n. These are particularly useful in computations, since only finite sums are needed to compute them.

As an example, let's take $\mathbf{h} = (6, 1, -1)$ (in our short vector notation). The LSI system $\mathbf{x} \to \mathbf{h} * \mathbf{x} = \mathbf{y}$ is given by the formula

$$y_n = 6x_n + x_{n-1} - x_{n-2}, \text{ for all } n. \tag{101}$$

See how there is only 3 terms in the sum given by the convolution with **h**.

This is an example of a *causal* system: the value of y_n depends only on the value of x_n and earlier coefficients in **x**. In general, a LSI system with be causal if the impulse response **h** is zero on the negative integers. We usually want to build causal filters.

For the example above, the Z transform is the polynomial

$$H(Z) = 6 + Z - Z^2. \tag{102}$$

Notice this polynomial factors, as $(6 + Z - Z^2) = (3 - Z)(2 + Z)$. The linear terms $(3 - Z), (2 + Z)$ are called couplets. From the couplets, we can see the zeros of this polynomial are $Z = 3$ and $Z = -2$. These zeros have magnitude bigger than one, so they live outside the unit circle in the complex plane. Because of this, we say the system is minimum phase.

By the Fundamental Theorem of Algebra, any polynomial can be factored into couplets. The zeros can be identified as points on the complex plane. If the zeros are all outside the unit circle, we say the system is minimum phase. If the zeros are inside the unit circle, we say the system is maximum phase. If some zeros are inside the circle, and some are outside, we say the system is mixed phase.

Minimum and maximum phase will have something to do with delays in our systems, which we will see later.

8 Lecture 8: Filters and Decimation

There are two key steps in the discrete wavelet transform: filtering (both lowpass and high pass) and decimation.

Filtering is done by a convolution filter. In the simplest case, we can use a simply averaging filter as a lowpass filter, and differencing as the high pass filter. For instance, we filter a sequence of data points $\mathbf{x}$ to obtain sequences $\mathbf{y}, \mathbf{z}$ as follows:

$$y_k = \frac{x_k + x_{k+1}}{2}, \qquad z_k = \frac{x_k - x_{k+1}}{2}. \tag{103}$$

Notice we get twice the data out as we get in. For instance, is $\mathbf{x}$ is a sequence of length 1024 data points, both $\mathbf{y}$ and $\mathbf{z}$ have about 1024 data points each, totally 2048 data points. Neither $\mathbf{y}$ nor $\mathbf{z}$ has enough information, but together they do, for we can write

$$y_k + z_k = \frac{x_k + x_{k+1}}{2} + \frac{x_k - x_{k+1}}{2} = x_k, y_k - z_k = \frac{x_k + x_{k+1}}{2} - \frac{x_k - x_{k+1}}{2} = x_{k+1}. \tag{104}$$

That is, the $\mathbf{y}$ and $\mathbf{z}$ can be used to compute the original sequence $\mathbf{x}$, so we have an invertible transformation.

Looking a bit more closely, we notice that if k is even, from $y_k + z_k$ and $y_k - z_k$ we recover both x_k (an even index) and x_{k+1} (an odd index). So from the even data in $\mathbf{y}, \mathbf{z}$, we can recover all the data in $\mathbf{x}$. These leads to the idea of *decimation*, where we throw away every other data point in $\mathbf{y}, \mathbf{z}$. We can do this, and still get an invertible operation.

Thus, our simple wavelet transform (the Haar transform) is defined by the convolutions

$$y_k = \frac{x_k + x_{k+1}}{2}, \qquad z_k = \frac{x_k - x_{k+1}}{2}, \qquad \text{for } k \text{ even only}, \tag{105}$$

and we ignore the odd indices. The convolutions are the filters; the throwing away of odd indices is the decimation.

I need to say something about the filter response of those guys. Blah blah blah, will add this later, bu th e point is

$$|H(\omega)| = \cos(\pi\omega), \qquad |G(\omega)| = \sin(\pi\omega), \qquad 0 \le \omega \le 1/2. \tag{106}$$

So the first is lowpass, second is high pass.

Review how the book defines lowpass, high pass.

The two convolution operators with coefficients $\mathbf{h} = (1/2, 1/2)$ and $\mathbf{g} = (1/2, -1/2)$ represent lowpass and high pass filters, respectively. To see this, we compute the Fourier series and simplify, to see that the lowpass filter has frequency response

$$\begin{aligned} H(\omega) &= \sum h_k e^{2\pi i k\omega} &(107)\\ &= (1/2)e^{2\pi i 0\omega} + (1/2)e^{2\pi i 1\omega} &(108)\\ &= e^{\pi i\omega}\frac{e^{-\pi i\omega} + e^{\pi i\omega}}{2} &(109)\\ &= e^{\pi i\omega}\cos(\pi\omega), &(110) \end{aligned}$$

while the highpass filter has response

$$\begin{aligned} G(\omega) &= \sum g_k e^{2\pi i k\omega} &(111)\\ &= (1/2)e^{2\pi i 0\omega} - (1/2)e^{2\pi i 1\omega} &(112)\\ &= e^{\pi i\omega}\frac{e^{-\pi i\omega} - e^{\pi i\omega}}{2} &(113)\\ &= -e^{\pi i\omega} i \sin(\pi\omega). &(114) \end{aligned}$$

Notice the little trick we used to factor out an exponential, to get a sum/difference of two exponentials that are complex conjugates of each other. Result is the simple trig functions, sin and cos. It's a good trick – remember it!

Taking absolute values, we see that the lowpass filter has magnitude response

$$|H(\omega)| = |\cos(\pi\omega)|, \tag{115}$$

while the highpass filter has magnitude response

$$|G(\omega)| = |\sin(\pi\omega)|. \tag{116}$$

The frequency response (on the normalized frequency range $0 \le \omega \le 1/2$) is shown in Figure 3.

It is also interesting to note the sum of the squares is a constant, so

$$|H(\omega)|^2 + |G(\omega)|^2 = \cos^2 + \sin^2 = 1. \tag{117}$$

We will see this again with the Daubechies wavelet transformation.

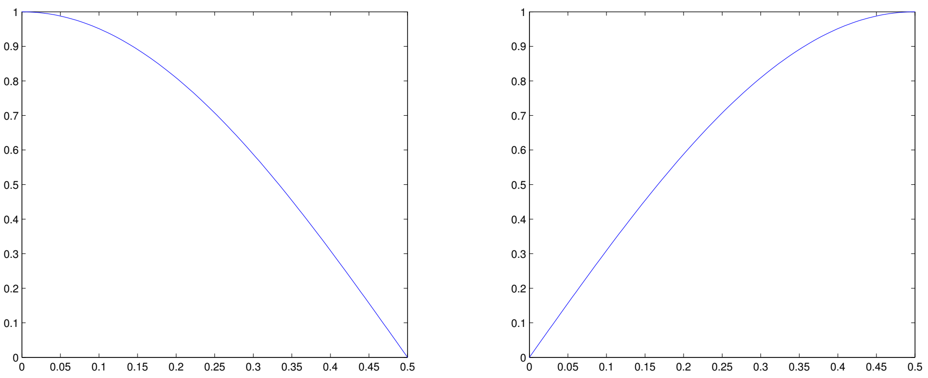


Figure 3: Magnitude response of the lowpass and highpass filters $H(\omega), G(\omega)$, respectively.

Now, we have a lot of freedom in selecting our low and high pass filters. The two given above, with the cos and sin frequency response, are okay, but not great. It might be useful to have filters with a sharper cutoff, as shown in the frequency response of the two filters in Figure 4. The point of the wavelet transform construction is that we can choose our lowpass, and high pass as we please, then decimate, to get (hopefully) an invertible transformation. And if we are really careful, we can get an orthogonal transformation.

The book is rather casual on what constitutes a lowpass or a highpass filter. For a lowpass, we would expect something like the magnitude response to be close to one for low frequencies, and close to 0 for high frequencies. The book defines the following condition

$$\text{Condition 5.1 (Lowpass)}| \qquad H(0)| = 1, \quad |H(1/2)| = 0. \tag{118}$$

For a high pass filter, we want the reverse:

$$\text{Condition 5.2 (Highpass)} \qquad |G(0)| = 0, \quad |G(1/2)| = 1. \tag{119}$$

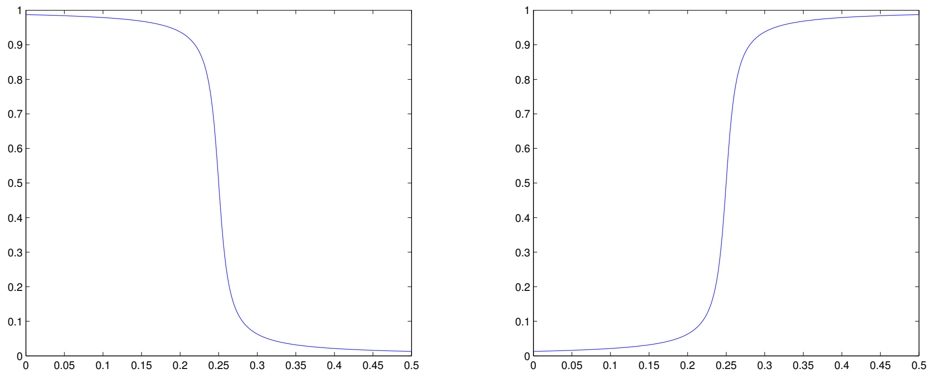


Figure 4: Magnitude response for better, sharper lowpass and highpass filters.

Since our magnitude response is always continuous (Why? Think about this!), these two conditions do give something like $H(\omega) \approx 1$ for ω near zero, and $H(\omega) \approx 0$ for ω near 1/2.

(Keep in mind I am using normalized frequency, so $\omega = 1/2$ corresponds to the frequency at 1/2 the sampling rate. The book would use $\omega = \pi$ which makes sense for a different normalization. I still think my way is better!)

9 Lecture 9: Haar Wavelet Transform

In the last section, we wrote out the simple Haar transformation using convolution and decimation, yielding the formulas:

$$y_k = \frac{x_k + x_{k+1}}{2}, \qquad z_k = \frac{x_k - x_{k+1}}{2}, \qquad \text{for } k \text{ even only.} \tag{120}$$

This formula can be applied to input sequences $\mathbf{x}$ of arbitrary length, but in practice we fix a length and think of this as a finite dimensional linear transformation.

It is convenient to organize this transform into a matrix, where the top half of the matrix gives the $\mathbf{y}$ and the bottom half gives the $\mathbf{z}$. In the 8x8 case, the matrix looks like this:

$$\frac{1}{2}\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix}. \tag{121}$$

It is an easy check to see that the rows are all orthogonal to each other. It is convenient to renormalize the matrix so it becomes orthonormal – that is, the rows will have length one if we write it as

$$W = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix}. \tag{122}$$

This is called the 8x8 Haar wavelet transformation matrix. Obviously we can extend this to any nxn dimension (n even) and obtain and orthogonal matrix. Convince yourself that we have the identity

$$W^*W = I. \tag{123}$$

9.1 Sample code

Let's try out the Haar wavelet transform on some real data, to see what it does. We will take a one period of the sine function, with 1024 sample points, and apply the Haar wavelet transform. In MATLAB, the code is

```
t = linspace(0,1,1024);
x = sin(2*pi*t);
y = HWT1D1(x);
plot(x), plot(t)
```

The result is shown in Figure 5. The plot on the right shows the output of the computation. Note it is in two parts: the first half looks like the original sine wave, the second half looks like something close to zero. We can get a better picture by plotting these two halves separately, as in Figure 6. Notice the second half looks like a cosine wave (the derivative of the original sine, because the high pass filter is a difference operator). But it is very small – the scaling factor shown in the plot is 10^{-3}.

So why do we do this? The point is to compress most of the energy in the signal into few coefficients. We can see this by plotting the cumulative energy of the original signal and of the transforms signal, as shown in Figure 7. What we notice is that the cumulative energy for the Haar transform version rises to one much faster. Which is to say, more energy is concentrated in fewer coefficients.

9.2 Iterating the transform

Let's be careful about what it means to iterate the Haar transform. You don't apply it over and over ageing to the whole output. What you do is apply the transform to the 1024 samples in the initial signal. Then you apply it to the 512 samples of the first half of the output – this is where the signal is large and we hope to compress it. The result is the output is 512 samples, the first

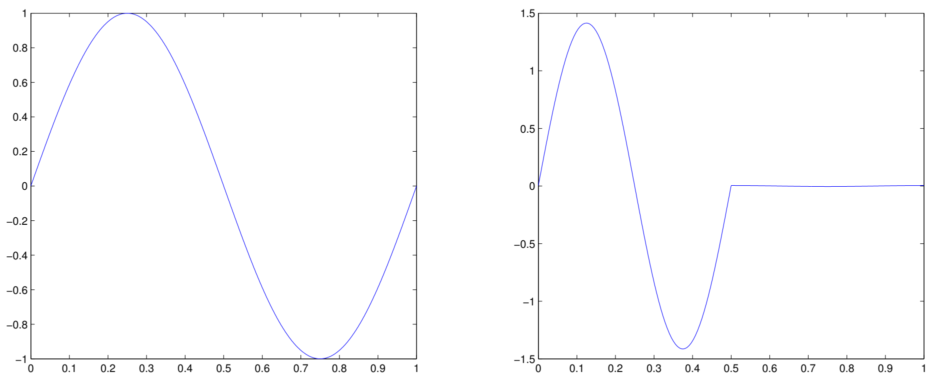


Figure 5: A sampled sine wave, and its Haar transform (notice low pass, high pass split).

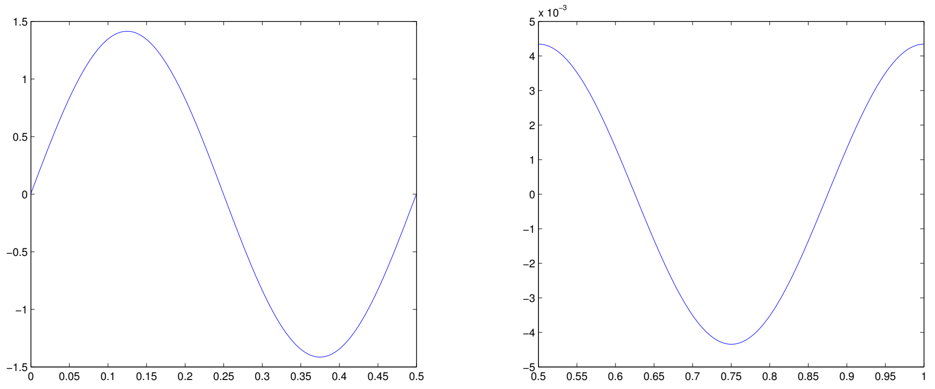


Figure 6: The two parts of the Haar transform, low pass on left, high pass on right.

256 are the result of a low pass, the second 256 are the result of the high pass. The remaining 512 sample are left untouched.

On the next iteration, we apply the transform to the first 256 samples, and leave the rest alone. We can plot the results of the three iterations in Figure 8. What you should notice is that the main bulk of the signal shows up in the first 1/2 of the 1st iteration, in the first 1/4 of the 2nd iteration, and in the first 1/8 of the 3rd iteration. Overall, this means that with more iterations, there are fewer "big coefficients."

Let's iterate the transform, and see what happens with the cumulative energy.

```
y1 = HWT1D(x,1);
y2 = HWT1D(x,2);
y3 = HWT1D(x,3);
plot(t,CE(x),t,CE(y1),t,CE(y2),t,CE(y3))
```

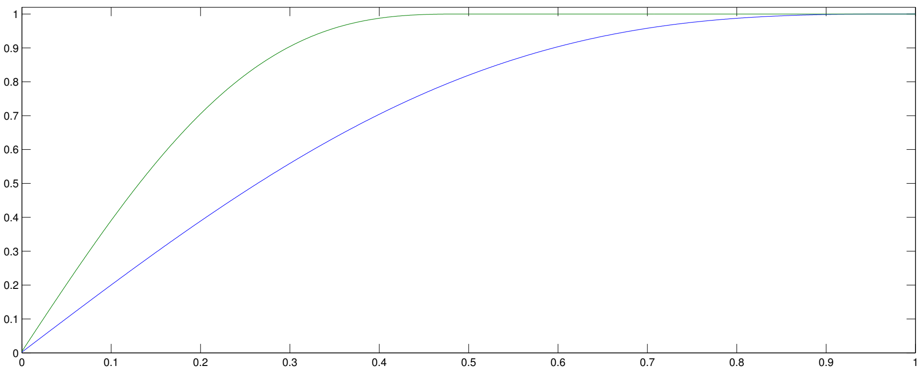


Figure 7: Cumulative energy of the original signal (sine) and its Haar transform.

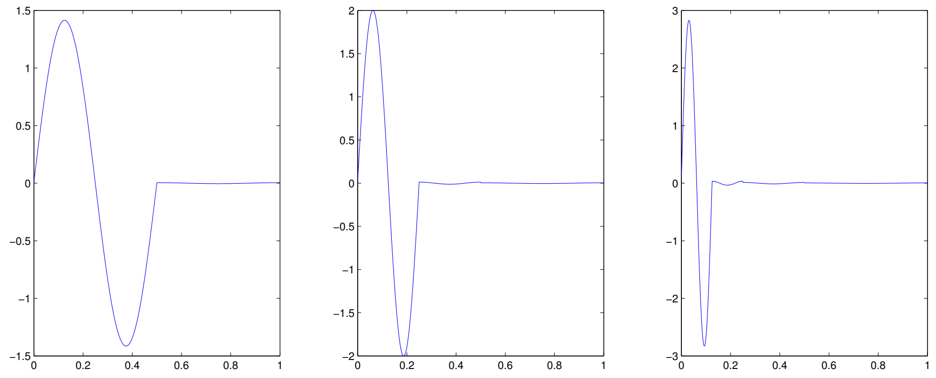


Figure 8: 1st, 2nd and 3rd iterations of the Haar transform, applied to the sine wave.

A plot of the various cumulative energies shows the more we iterate the Haar transform, the more energy gets concentrated into fewer coefficients. As shown in Figure 9.

9.3 Operations count

It is worth noting that the above 8x8 matrix, when applied to vector **x**, will compute the output in about 16 floating point operations. There are 4 additions, 4 subtractions, and 8 multiplications by $1/\sqrt{2}$. In the general nxn case, it will take 2n operations. This is considerably faster than the usual matrix-vector multiplication, which takes $O(n^2)$ operations.

When you iterate the Haar wavelet transform, it is not much worse. The reason is we decrease the size of the input vector by a factor of 2 at each stage. So, for instance, if we start with a vector of length 1024, we need 2*1024 operations. In next iteration, we have an input of length 512, operations needed are 2*512. Next step, input is length 256, operations is 2*256. This continues

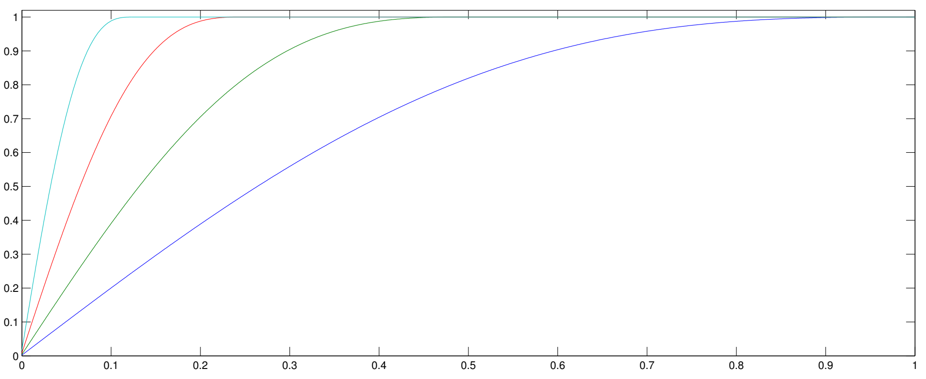


Figure 9: Cumulative energy of the original signal, and 3 iterations of the Haar transform.

until we get down to an input of length 1. The total operations is

$$2 \cdot 1024 + 2 \cdot 512 + 2 \cdot 256 + 2 \cdot 128 + \cdots 2 \cdot 1 = 2 \cdot 2047, \tag{124}$$

or about $4n$ operations, with $n = 1024$ the length of the original input.

With other wavelet transforms we will consider in later lecture, there are typically k operations per output, where k is a small number like 2,4,6, 8, or so. The same calculation as above shows the total number of operations for an input vector of length n is about $2kn$.

9.4 Ordering the iterations

In practice, it has been found useful to iterate always on the lowpass output. However, this is not a requirement of the wavelet transform. One could iterate on the high pass output only. Or one could alternate between the lowpass output, and the high pass output. At each stage, it is important to choose only one of the lowpass or high pass output, so that the total number of operations to compute is kept low.

The choice can be made adaptively – you check the total energy in the low and high pass output, and iterate on the one that has more energy (on the theory that this is where you get the most gains).

Or, you could choose to iterate on both the lowpass and high pass outputs. This process is known as using “wavelet packets” or optimal sub band tree structuring.

But for now, let’s just assume we iterating on the low pass outputs only.

10 Lecture 10: Daubechies Wavelet Transform

Ingrid Daubechies is a professor at Princeton. Her key idea was to worry less about the details of the filter response, and worry more about the mechanics of the computation. In particular, she found it was better to work directly with the filter coefficients that appear in the low and high pass filter of the transform.

10.1 Daubechies 4

So, for instance, in the Haar transform we have filter coefficients $(1/2, 1/2)$ and $(1/2, -1/2)$, we can instead imagine filters with four coefficients (h_0, h_1, h_2, h_3) and (g_0, g_1, g_2, g_3) and do the same construction of a transform matrix. Then ask: what conditions on the $h's$ and $g's$ give us an orthogonal matrix.

Let's do the 8x8 case. For these short, length-four convolution filters, we get a matrix like this:

$$W = \begin{bmatrix} h_3 & h_2 & h_1 & h_0 & 0 & 0 & 0 & 0 \\ 0 & 0 & h_3 & h_2 & h_1 & h_0 & 0 & 0 \\ 0 & 0 & 0 & 0 & h_3 & h_2 & h_1 & h_0 \\ h_1 & h_0 & 0 & 0 & 0 & 0 & h_3 & h_2 \\ g_3 & g_2 & g_1 & g_0 & 0 & 0 & 0 & 0 \\ 0 & 0 & g_3 & g_2 & g_1 & g_0 & 0 & 0 \\ 0 & 0 & 0 & 0 & g_3 & g_2 & g_1 & g_0 \\ g_1 & g_0 & 0 & 0 & 0 & 0 & g_3 & g_2 \end{bmatrix}. \tag{125}$$

Remember, like with the Haar matrix, we start with a Toeplitz matrix (constant along diagonals), and throw away every other row. That's why the h-values look like they get shifted by 2 as you go down by rows. Same for the g's. Now, half way through the matrix, at the end of the h-rows. we run out of space for the $h's$, so we wrap around back to the first to columns. Similar thing at the end of the g-rows: just wrap around.

Now, here's a neat trick. To get the h-rows perpendicular to the g-rows, just force the g's to be the same values as the h's, except in reverse order. That is, we write

$$(g_0, g_1, g_2, g_3) = (h_3, -h_2, h_1, -h_0). \tag{126}$$

For if we do that, the dot product is zero:

$$(h_0, h_1, h_2, h_3) \cdot (g_0, g_1, g_2, g_3) = (h_0, h_1, h_2, h_3) \cdot (h_3, -h_2, h_1, -h_0) = h_0 h_3 - h_1 h_2 + h_2 h_1 - h_3 h_0 = 0. \tag{127}$$

To get the first two rows orthogonal, we need that

$$(h_3, h_2, h_1, h_0, 0, 0, 0, 0) \cdot (0, 0, h_3, h_2, h_1, h_0, 0, 0) = h_0 h_2 + h_1 h_3 = 0. \tag{128}$$

By symmetry in the matrix, once the first two rows are orthogonal, they all are.

We also want the rows to have unit length: that is easily satisfied by requiring

$$h_0^2 + h_1^2 + h_2^2 + h_3^2 = 1. \tag{129}$$

Now, the h's should give us a low pass filter – in particular, they need to "kill" the high frequencies, so we should require that their Fourier transform has a zero at Nyquist frequency $\omega = 1/2$. Thus we require

$$0 = H(1/2) = h_0 e^{2\pi i 0\cdot(1/2)} + h_1 e^{2\pi i 1\cdot(1/2)} + h_2 e^{2\pi i 2\cdot(1/2)} + h_3 e^{2\pi i 2\cdot(1/2)}. \tag{130}$$

Simplified, this just reads

$$0 = h_0 - h_1 + h_2 - h_3. \tag{131}$$

A careful count of the above considerations shows we have 3 equations in the 4 unknowns h_0, h_1, h_2, h_3, while the g's are completely determined by the h's. Let's add one more equation to get a solvable system: we require that the derivative of $H(\omega)$ is also zero at $\omega = 1/2$.. Thus

$$0 = H'(1/2) = h_0 \cdot 0 e^{2\pi i 0\cdot(1/2)} + h_1 \cdot 1 e^{2\pi i 1\cdot(1/2)} + h_2 \cdot 2 e^{2\pi i 2\cdot(1/2)} + h_3 \cdot e^{2\pi i 2\cdot(1/2)}. \tag{132}$$

Simplified, we have

$$0 = h_1 - 2h_2 + 3h_3. \tag{133}$$

Thus, the 4 equations in 4 unknowns are

$$\begin{aligned} h_0^2 + h_1^2 + h_2^2 + h_3^2 &= 1 \qquad (134)\\ h_0 h_2 + h_1 h_3 &= 0 \qquad (135)\\ h_0 - h_1 + h_2 - h_3 &= 0 \qquad (136)\\ h_1 - 2h_2 + 3h_3 &= 0. \qquad (137) \end{aligned}$$

. A solution is given by

$$\begin{aligned} h_0 &= \frac{1}{4\sqrt{2}}(1+\sqrt{3}) \qquad (138)\\ h_1 &= \frac{1}{4\sqrt{2}}(3+\sqrt{3}) \qquad (139)\\ h_2 &= \frac{1}{4\sqrt{2}}(3-\sqrt{3}) \qquad (140)\\ h_3 &= \frac{1}{4\sqrt{2}}(1-\sqrt{3}). \qquad (141) \end{aligned}$$

As an exercise, check that this really is a solution. If you put all the h's with opposite sign, you get another solution. Are there others??

We can plot the frequency response of the filter given by the h's, to verify it is a low pass filter. You should also plot the frequency response for the corresponding g's, to verify that it is a high pass filter. You might even like to verify that the resulting matrix W really is orthogonal. Using MATLAB, check that

$$W^* W = I. \tag{142}$$

It's not too hard to see how to generalize this to any nxn matrix, so long as n is even.

This matrix transform is called the Daubechies-4 wavelet transform.

10.2 Odd Daubcechies

Ask yourself, why didn't we use a filter of length 3? Turns out it can't work, for trivial reasons. Try it out. The general Daubechies construction only works for even length filters.

10.3 Daubechies 6

We can do the same construction with filters of length six. The 6 equations obtained are

$$\begin{aligned}
h_0^2+h_1^2+h_2^2+h_3^2+h_4^2+h_5^2 &= 1 && (143)\\
h_0h_2+h_1h_3+h_2h_4+h_3h_5 &= 0 && (144)\\
h_0h_4+h_1h_5 &= 0 && (145)\\
h_0-h_1+h_2-h_3+h_4-h_5 &= 0 && (146)\\
h_1-2h_2+3h_3-4h_4+5h_5 &= 0 && (147)\\
h_1-4h_2+9h_3-16h_4+25h_5 &= 0. && (148)
\end{aligned}$$

Where do these equations come from? Well, the first says we want rows of unit magnitude. The second says we need rows 1 and 2 to be perpendicular to each other. The third says we want rows 1 nd 3 to be perpendicular to each other. The fourth, fifth and sixth are requirements on the frequency response, namely that $H(\omega)$ and its first two derivatives should be zero at Nyquist $\omega = 1/2$. That is,

$$H(1/2) = H'(1/2) = H''(1/2) = 0. \qquad (149)$$

Six equations in six unknowns, turns out we can solve this. Done. That gives the Daubechies 6 wavelet transformation.

What about g. Well, same trick as before: g is the reverse of h, with alternating signs.

10.4 Daubechies 2N

In general, we can do this for filters of even length 2N. Get 2N equations, in 2N unknowns (the h's). These have been solved numerically for reasonable value sod N. You can find them in MATLAB, or anywhere on the web.

We plot the low pass filter response for the five different Daubechies wavelet transforms, shown in Figure 10. Notice the higher order transforms give a sharper cutoff in the frequency response.

It is worth noting that the textbook's code gives slightly different Daubechies coefficients than what the MATLAB wavelet toolkit gives. In the book, $H(0) = \sqrt{2}$, while in MATLAB, one gets

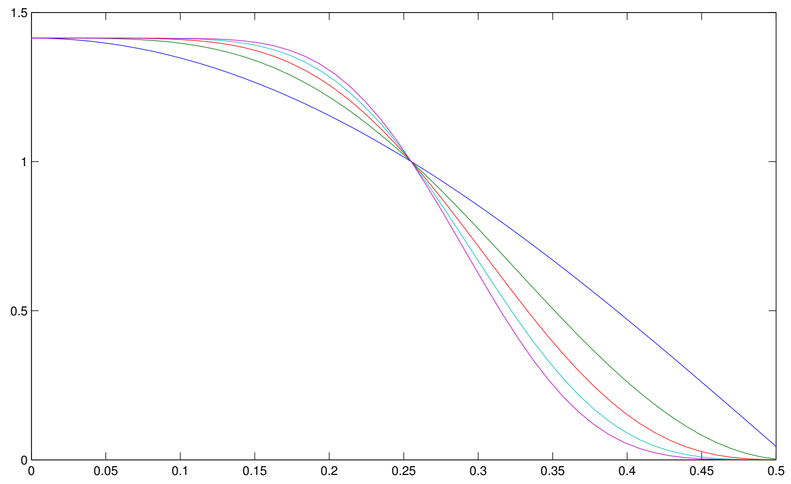


Figure 10: Low pass filter response for Daubechies transforms of length 2,4,6,8,10.

$H(0) = 1$. These are just a different normalization, but you should be aware that there is a difference.

11 Lecture 11: Wavelet transforms of images

A quick and dirty way to do a 2D transform is to apply a 1D transform twice, once in the horizontal direction, and then again in the vertical direction. Not always the best thing to do, but it is a start.

For a 2D image in black and white, you can represent the image as a matrix of row and columns, with each entry in the matrix corresponding to the intensity at the aligned pixel location. To apply the Haar wavelet transform in 2D, you first apply it to every row in the matrix, and then on the resulting output, apply it to each column.

To iterate, you locate one corner of the matrix corresponding to the low pass output (in both horizontal and vertical) and apply the transform just to that corner. Wash, rinse, and repeat.

11.1 Some sample transforms

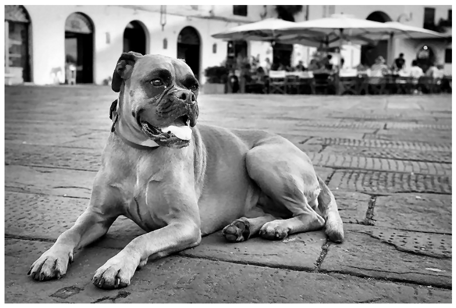
We can load in some black and white images and apply the Haar wavelet transform to both rows and columns – this give a 2D transform. Some code:

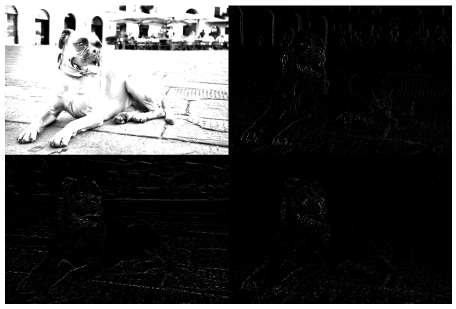
```
gry=ImageNames('ImageType','Grayscale');
file=gry{7};
```

```
A=ImageRead(file);
ImagePlot(A)
B = HWT2D1(A);
ImagePlot(B)
C = HWT2D(A,2);
ImagePlot(C)
```

The matrix A is the original image of a dog, matrix B is the first iteration of the Haar transform, and C is the 2nd iteration. These are displayed in Figure 11. It is interesting to note that in the first iteration, the lowpass filtered version fits in the upper left corner of the output, taking up 25% of the data space. The high pass outputs are in the other corners, and contain much less energy of the data (which is why it looks so black). You should try this code yourself and see what is sitting in those high pass output images – you will see some of them reveal vertical features in the original image, others reveal horizontal features. This is a very crude, but fast way of doing edge detection.

When the algorithm iterates, it only acts on the low pass "corner" of the transformed image, and leaves the other 3 corners the same. You can see the top left corner in the first iteration gets shrunk to a smaller corner in the next iteration. For the rest, it is hard to see in these images (everything is close to black), so you should try the code yourself. Or read the code provided by the textbook author.





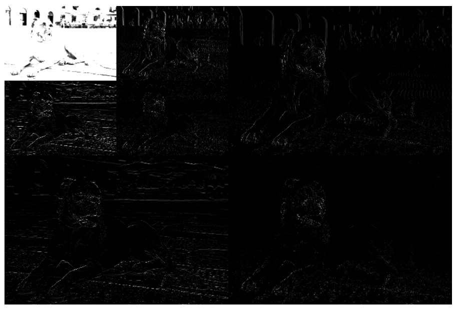


Figure 11: An image, and 2 iterations of the Haar transform in 2D.

Under the philosophy that we do transforms to concentrate energy, we can plot the cumulative energy for the image and its two transforms. This is shown in Figure 12. What you should notice is that the CE for the transformed data rises to 1 much more quickly. Indeed, in the 2nd iteration, we get 95% of the energy in just the first 5% of the coefficients. For the original image, it takes about 80% of the coefficients to meet that same level of energy. Which is to say, there is a lot more opportunity to compress in the transformed data.

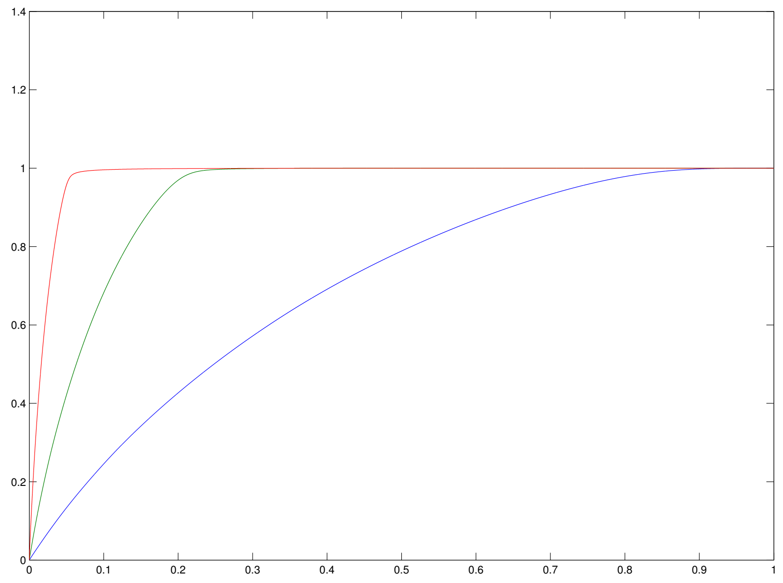


Figure 12: The cumulative energy for the image, and 2 iterations of the Haar transform.

11.2 Operations count

Try to Figure this out. Take a $m \times n$ matrix. Each row has n entries, so applying the 1D Haar transform to 1 row takes 2n operations. To do all the rows takes 2mn operations. Applying to the resulting columns takes another 2mn operations, so total is 4mn operations for one step of the Haar transform.

When iterating, we only apply to the corner, which is size $\frac{m}{2} \times \frac{n}{2}$. Thus we only need 4 times $mn/4$ operations to get the second iteration.

To do all the iterations takes a total of

$$4mn(1 + \frac{1}{4} + \frac{1}{16} + \cdots) = 4mn\frac{1}{1 - 1/4} = 16mn/3, \qquad (150)$$

or roughly $5mn$. Anyhow, it is linear in the size of the original data, which again indicates that this is a very fast transform.

Try to figure out how many operations are needed when using the Daubechies wavelet transform in 2D. Again, you should find that this is also a fast transform.

APPENDIX 1: Collected wisdom

In case we are missing the forest for the trees, here are some basic facts about signal processing that you should always keep in mind. (I will add to this as the course goes along.)

Signals:

- Signals are functions (of time, space, etc).
- Many useful signals are sums of sines and cosines.
- A sine wave or cosine wave is specified by a frequency and an amplitude. Negative frequencies give basically the same signal as a positive frequency.
- Sines, cosines are the same function, just shifted in time.
- Unless we know the start time, sine waves and cosine waves at the same frequency are basically the same thing. Same for any shifts of these waves.
- The complex sinusoid $e^{2\pi i\omega t}$ makes the algebra of sines and cosines easy. But in real life, we never really see complex valued waves.

Physical signals:

- We hear sounds in the range 20Hz to 20,000 Hz.
- Seismic waves are measured in the range 4Hz to 150 Hz (approx.) As technology improves, we expand this range.
- Radio waves are in the range of kilohertz (AM), megahertz (FM), gigahertz (cellphones).
- Sound waves are acoustic waves (variations in air pressure). Seismic waves are elastic waves (solid motion). Radio waves are electromagnetic waves.

Sampled signals:

- For practical reasons, we sample signals.
- Sample rate determines the highest frequency we can represent

$$\frac{1}{2}(\text{sample rate}) = \text{Nyquist rate.} \tag{151}$$

- Frequencies higher than that are aliases, and cause errors.
- In real systems, we use electronics to eliminate those higher frequencies, before sampling.

Systems:

- Basic mode is “Signal in $\to$ Signal out.”
- Convolution gives a LSI system, and vice versa.
- Specify a LSI system by its impulse response $\mathbf{h}$.
- Practical LSI system given by finite $\mathbf{h}$ or ratio of two such finite ones.

APPENDIX 2: Eigenvalues and eigenvectors

Recall a matrix A has an eigenvector $\mathbf{x} \neq 0$ and an eigenvalue λ if they satisfy

$$A\mathbf{x} = \lambda \mathbf{x}. \tag{152}$$

That is, the matrix applied to vector $\mathbf{x}$ just returns the same vector $\mathbf{x}$, multiplied by the number λ.

As an example, with the matrix $A = \begin{bmatrix} 2 & 4 \\ 4 & 2 \end{bmatrix}$, we find two eigenvectors $[1, 1]$ and $[1, -1]$ by observing

$$\begin{aligned} \begin{bmatrix} 2 & 4 \\ 4 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} &= 6 \begin{bmatrix} 1 \\ 1 \end{bmatrix} \\ \begin{bmatrix} 2 & 4 \\ 4 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \end{bmatrix} &= -2 \begin{bmatrix} 1 \\ -1 \end{bmatrix}. \end{aligned}$$

So 6 and -2 are the eigenvalues of the matrix. Notice that $6 - 2 = 4$, the trace of the matrix (sum of the diagonal elements), while $6 * (-2) = -12$, the determinate of the matrix. This is a general property of the eigenvalues: they tell us a lot about the matrix.

In the frequency response, the only difference is that we are working in infinite dimensions. The eigenvectors are signals $\mathbf{x}$ that are complex exponentials at frequency ω. The corresponding eigenvalues are the complex numbers $H(\omega)$. That is, when $\mathbf{x}$ is a complex exponential, we have the eigenvector equation

$$\mathbf{h} * \mathbf{x} = \lambda \mathbf{x}, \tag{153}$$

where the eigenvalue λ is the (complex) number $\lambda = H(\omega)$, given by the frequency response evaluated at ω.

APPENDIX 3: Music and frequencies

Humans can hear sounds in the range of 20Hz to 20,000 Hz. (Hz = Hertz = cycles per second). Basically any periodic wave that repeats itself that many times a second will be heard by the human ear as a musical tone. The frequency (cycles per second) corresponds to pitch of the note. The low (deep) sounds have the low frequency (eg 25) and the high (shrill) sounds have the high frequency

A piano plays notes in the range of about 25 Hz to 4200 Hz. The note "A above middle C" is defined as a pitch of 440 Hz. All the other pitches are defined relative to that frequency. The equal tempered scale uses powers of the number $\alpha = \sqrt[12]{2}$ to define other frequencies as

$$freq = 440 * \alpha^n, \tag{154}$$

where n is the number of semitones above "A", or below it for n negative.

Doubling the frequency ($n = 12$ semitones up) corresponds to moving up an octave in the musical scale. Halving the frequency ($n = -12$ semitones down) moves down an octave.

Bach was involved in the notion of an equal tempered scale, which uses the power of the 12th root of 2 to determine frequencies. You may be familiar with his keyboard composition call"The Well-tempered Klavier." This was motivated by the desire to get all instruments in an orchestra to sound in tune, not matter what key they were playing in.

Before equal tempering, notes were based on the idea that frequencies that were related as the ratio of simple fractions often sound good together. This is deeply connected to the notion of harmonics in musical tones, but it seems rather mathematical.

For instance, the frequencies 440Hz, 550Hz, 660Hz all sound good together, and their ratios are

$$\frac{550}{440} = \frac{5}{4} \qquad \frac{660}{440} = \frac{3}{2}, \tag{155}$$

which are simple fraction with small integers in the fractions. These three notes correspond to the three notes in a major triad (A, C#, E). It is interesting to note that the middle note, 550Hz, would actually be as high as 554Hz in the equal tempered scale. To those of you with good ears, the 554Hz tone sounds a little sharp. We get this 554Hz value by going up four semitones in the equal tempered scale, so $440 * 2^{4/12} = 554.37$.

You can experiment with these ideas in MATLAB by setting up some simple signals and playing them out.

```
Fs = 10000; % the sampling rate
dt = 1/Fs; % the time step
T1 = 0:dt:1; % one second of time, in steps of dt
A = sin(2*pi*440*T1);  % the note A above middle C
Cs = sin(2*pi*550*T1); % the note C#
E = sin(2*pi*660*T1); % the note E
sound([A,Cs,E],Fs);  % play the notes one after the other
sound(.3*(A+Cs+E),Fs); % play the notes all together as one
```

If you are getting tired of hearing pretty sine waves, try raising it to some (odd) power, to get some richer harmonics in your music.

```
Fs = 10000;
```

```
dt = 1/Fs;
T1 = 0:dt:1;
A = sin(2*pi*440*T1).^5;  % the note A above middle C, with harmonics
Cs = sin(2*pi*550*T1).^5; % the note C#
E = sin(2*pi*660*T1).^5; % the note E
sound([A,Cs,E],Fs);
sound(.3*(A+Cs+E),Fs);
```

To get the major scale, using fractions instead of powers of two, you can use the ratios

$$\frac{1}{1}, \frac{9}{8}, \frac{5}{4}, \frac{4}{3}, \frac{3}{2}, \frac{27}{16}, \frac{15}{8}, \frac{2}{1}. \tag{156}$$

In the A major scale, this corresponds to frequencies

$$440, 495, 550, 586.7, 660, 742.5, 825, 880\, Hz. \tag{157}$$

Some might argue that other fractions are better. For instance, we might replace the 5/4 with $81/64 = (1/4) * (3/2)^4$. If you know something about music theory, you will see this is connected to the circle of fifths idea.

Now, just because its fun, try this. It is a sweep through the frequency range

```
Fs = 10000;
dt = 1/Fs;
T1 = 0:dt:1;
sweep = sin(2*pi*440*T1.^3);  % a sweep
sound(sweep,Fs);
```