Linear Algebra Review
Daniel Kersten

Vector operations

■ Dimension of a vector.

You can get the dimensionality of a vector using **Dimensions[]**, or **Length[]**.

In[16]:=
```
v = {2.1, 3, -0.45, 4.9};

Dimensions[v]
```

Out[17]=
```
{4}
```

Dimensions[], will give you the dimensions of a matrix, while **Length[]** tells you the number of elements in the list. For example,

In[18]:=
```
M = {{2,4,2}, {1,6,4}};
```

In[19]:=
```
Length[M]
```

Out[19]=
```
2
```

Try comparing **Length[M]** with **Dimensions[M]**.

■ Transpose of a vector.

The transpose of a column vector is just the same vector arranged in a row. However, because of the way Mathematica uses lists to represent vectors you don't have to distinguish between row and column vectors. In standard math notation, transpose of a vector **x**, is often written $\mathbf{x}^{\mathbf{T}}$. You can see a vector in column form by typing **v//MatrixForm**, or**:**

In[20]:=
```
MatrixForm[v]
```

Out[20]//MatrixForm=

$$\begin{pmatrix} 2.1 \\ 3 \\ -0.45 \\ 4.9 \end{pmatrix}$$

■ **Vector addition** is accomplished by simply adding the components of each vector to make a new vector. Note that the vectors all have the same dimension.

In[21]:=
```
a = {3,1,2};
b = {2,4,8};
c = a + b
```

Out[23]=
```
{5, 5, 10}
```

Vectors can be multiplied by a constant. We saw an example of this earlier.

In[24]:=
```
2 a
```

Out[24]=
```
{6, 2, 4}
```

■ **Metric length of a vector.**

It is unfortunate terminology, but **Length[]** does NOT give you the metrical length of the vector. In order to get the length of a vector, you calculate the Euclidean distance from the origin to the end-point of the vector. We get this by squaring each component, adding up the squares, and taking the square root. First, we will do this using the **Apply[]** function, where the **Plus** operation is applied to all the elements of the list. Note that the operation of exponentiation is "listable", that is it is applied to each element of the vector:

In[25]:=
```
a^2
```

Out[25]=
```
{9, 1, 4}
```

What is **a a** ?

In[26]:=
```
N[Sqrt[Apply[Plus, a^2]]]
```

Out[26]=
```
3.74166
```

If you wish, you can define your own function to apply to the list. What we have just calculated is the square root of the dot product or inner product of **a** with itself. The length of a vector **a** is often written as **|a|** in standard math notation. In the next section, we use the inner or dot product to calculate the metric length of a vector.

■ **Inner product**. To calculate the inner product of two vectors, you multiply the corresponding components and add them up:

In[27]:=
```
u = {u1,u2,u3,u4};
v = {v1,v2,v3,v4};
u.v
```

Out[29]=
```
u1 v1 + u2 v2 + u3 v3 + u4 v4
```

The **inner product** is also called the **dot product**. Later we will see what is meant by **outer product**. The inner product between two vectors **a** and **b** is written either as:

a.b or **[a,b]**, or $\mathbf{a}^{\mathbf{T}}\mathbf{b}$

Mathematica uses the dot notation.

One use of the inner product is to calculate the length of a vector. **a.a** is just the sum of the squares of the elements of **a**, so gives us another way of calculating the length of a vector.

In[30]:=
```
N[Sqrt[a.a]]
```

Out[30]=
```
3.74166
```

Let's define a function that will return the length of a vector, x:

In[31]:=
```
Vectorlength[x_] := N[Sqrt[x.x]]
```

■ **Projection.** The dot product, **a.b**, is equal to:

|a| |b| cos(angle between **a** and **b**)

In problem set 1, you calculate the output of a linear neuron model as the dot product between an input vector and a weight vector. Both the weight and input lists can be thought of as vectors in an n-dimensional space. Suppose the weight vector has unit length. Recall that you can normalize any vector to unit length by dividing by its length:

In[32]:=
```
v = v/Sqrt[v.v] ;
```

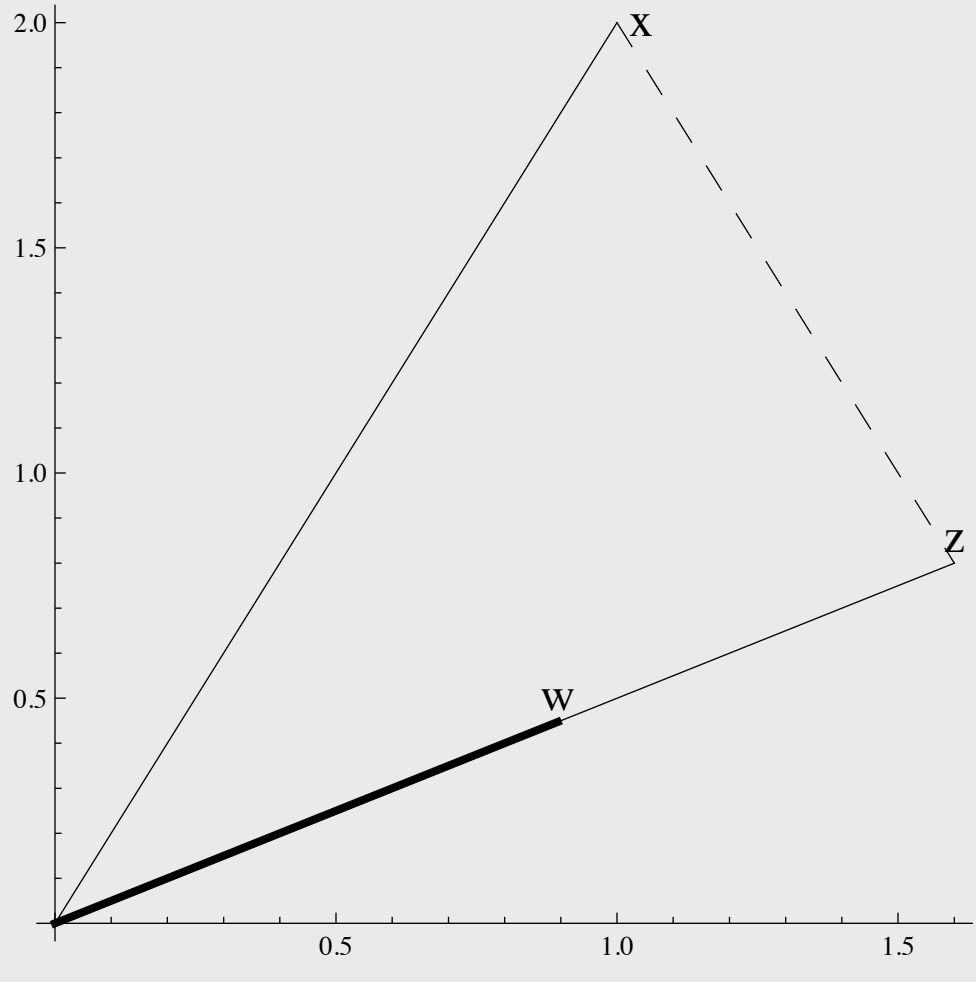
Geometrically, we can think of the output of a neuron as the projection of the activity of the neuron input activity vector onto the weight vector direction. Suppose the input vector is already perpendicular to the weight vector, then the output of the neuron is zero, because the cosine of 90 degrees is zero. As you found or will find with the cross-correlator of Problem Set 1, the further the input pattern is away from the weight vector, as measured by the cosine between them, the poorer the "match" between input and weight vectors, and the lower the response.

Here are three lines of code that calculate the two-dimensional vector **z** in the direction of **w**, with a length determined by "how much of **x** projects in the **w** direction":

In[33]:=
```
x = {1,2};
w = N[{2/Sqrt[5],1/Sqrt[5]}];
z = (x.w) w ;
```

In[36]:=
```
Show[Graphics[{Line[{{0, 0}, x}], Line[{{0, 0}, z}],
   {Dashing[{0.03`, 0.03`}], Line[{x, z}]},
   Text[Style["w", {"Helvetica-Bold", 18}], w, {0, -1}],
   Text[Style["x", {"Helvetica-Bold", 18}], x, {-2, 0}],
   Text[Style["z", {"Helvetica-Bold", 18}], z, {0, -1}],
   {AbsoluteThickness[3], Line[{{0, 0}, w}]}}], Axes → True,
 AspectRatio → 1]
```

Out[36]=



■ Angle between two vectors and orthogonality: Similarity measure between patterns

Often we will want some measure of the similarity between two patterns of neural firings. As we have just seen, one measure of comparison is the degree to which the two state vectors point in the same direction. The cosine of the angle between two vectors is one possible measure:

In[37]:=
```
cosine[x_,y_] := x.y/(Vectorlength[x] Vectorlength[y])
cosine[a,b]
```

Out[38]=
```
0.758175
```

Note that if two vectors point in the same direction, the cosine of the angle between them is 1:

In[39]:=
```
a = {2,1,3,6};
b = {6, 3, 9, 18};
cosine[a,b]
```

Out[41]=
```
1.
```

Try verifying that **w** and **z** from the previous section point in the same direction.

If two vectors point in the opposite directions, the cosine of the angle between them is -1:

In[42]:=
```
a = {-2,-1,-3,-6};
b = {6, 3, 9, 18};
cosine[a,b]
```

Out[44]=
```
−1.
```

Two vectors may point in the same direction, but could be quite different because they have different lengths. Another measure of similarity is the length of the difference between two vectors:

In[45]:=
```
Vectorlength[a - b]
```

Out[45]=
```
28.2843
```

■ **Orthogonality.** The case where vectors are at right angles to each other is an important special case that is worth spending a little time on. Consider an 8-dimensional space. One very familiar set of orthogonal vectors is the following:

In[46]:=
```
u1 = {1,0,0,0,0,0,0,0};
u2 = {0,1,0,0,0,0,0,0};
u3 = {0,0,1,0,0,0,0,0};
u4 = {0,0,0,1,0,0,0,0};
u5 = {0,0,0,0,1,0,0,0};
u6 = {0,0,0,0,0,1,0,0};
u7 = {0,0,0,0,0,0,1,0};
u8 = {0,0,0,0,0,0,0,1};
```

Each vector has unit length, and it is easy to see just by inspection that the inner product between any two is zero. On the

other hand, here is another set of 8 vectors in 8-space for which it is not immediately obvious that they are all orthogonal. These vectors are called Walsh functions:

In[54]:=
```
v1 = {1, 1, 1, 1, 1, 1, 1, 1};
v2 = {1,-1,-1, 1, 1,-1,-1, 1};
v3 = {1, 1,-1,-1,-1,-1, 1, 1};
v4 = {1,-1, 1,-1,-1, 1,-1, 1};
v5 = {1, 1, 1, 1,-1,-1,-1,-1};
v6 = {1,-1,-1, 1,-1, 1, 1,-1};
v7 = {1, 1,-1,-1, 1, 1,-1,-1};
v8 = {1,-1, 1,-1, 1,-1, 1,-1};
```

You can calculate the inner products between any two, and you will find out that they are all zero. Note that with the first set of vectors, $\{\mathbf{u}_i\}$, you can tell which vector it is just by looking for where the 1 is. For the second set, $\{\mathbf{v}_i\}$, you can't tell by looking at just one component. For example, the first component of all of the Walsh functions has a 1. You have to look at the pattern to tell which Walsh function you are looking at.

Suppose for the moment that we want to assign meaning to each of the patterns--each pattern is a code for some thing, like "grandma Tompkins", "grandma Wilke", and so forth. If we use the **u**'s, then we could look for the one neuron that lights up to find out which grandma it is representing--then neuron activity represented, for example, by the third element of the pattern could mean "grandma Wilke". This strategy wouldn't work if we encoded a collection of grandmothers using the **v**'s. The **v**'s give us a simple example of what is sometimes referred to as a **distributed code**. The **w**'s are examples of a **grandmother cell code**. The reason for this obscure terminology can be traced to earlier debates on whether there may be single cells in the brain whose firing uniquely determines the recognition of one's grandmother.

- **Orthonormality.** The Walsh set is orthogonal, but they are not of unit length. We have already seen some of the advantages of working with unit length vectors. The general issue of normalization comes up all the time in neural networks both in terms of limiting overall neural activity, and limiting synaptic weights. So it is sometimes convenient to normalize an orthogonal set, producing what is known as an *orthonormal* set of vectors:

In[62]:=
```
w1 = v1/Vectorlength[v1];
w2 = v2/Vectorlength[v2];
w3 = v3/Vectorlength[v3];
w4 = v4/Vectorlength[v4];
w5 = v5/Vectorlength[v5];
w6 = v6/Vectorlength[v6];
w7 = v7/Vectorlength[v7];
w8 = v8/Vectorlength[v8];
```

Vector representations, linear algebra

The issue of how information is to be represented is fundamental in the information sciences generally, as well as for neural network theory. A pattern of activity over a set of neurons is presumed to mean something, and there are different ways of coding the same meaning. But different codes have different properties. A code may not be sufficient to uniquely code all the possible things we need to represent. A code could be redundant and have more than one way of representing the same thing. This section continues with our review of the basics of vector and linear algebra by going a little more deeply into the subject. The pay-off will be some mathematics that provides intuition about issues of neural representation. You can think of this as a first lesson in the "psychology of linear algebra".

■ Basis sets

It is pretty clear that given any vector whatsoever in 8-space, you can specify how much of it gets projected in each of the eight directions specified by the unit vectors v1, v2, ...v8. But you can also build back up an arbitrary vector by adding up all the contributions from each of the component vectors. This is a consequence of vector addition and can be easily seen to be true in 2 dimensions. We can verify it ourselves. Pick an arbitrary vector g, project it onto each of the basis vectors, and then add them back up again:

In[70]:=
```
g = {2,6,1,7,11,4,13, 29} ;
```

In[71]:=
```
(g.u1) u1  + (g.u2) u2  +(g.u3) u3  +(g.u4) u4  +
(g.u5) u5  +(g.u6) u6  +(g.u7) u7  +(g.u8) u8
```

Out[71]=
```
{2, 6, 1, 7, 11, 4, 13, 29}
```

■ Exercise

What happens if you project **g** onto the normalized Walsh basis set defined by **{w1,w2,...}** above, and then add up all 8 components?

In[72]:=
```
(g.w1) w1  + (g.w2) w2  +(g.w3) w3  +(g.w4) w4  +
(g.w5) w5  +(g.w6) w6  +(g.w7) w7  +(g.w8) w8
```

Out[72]=
```
{2., 6., 1., 7., 11., 4., 13., 29.}
```

The projections, $\mathbf{g.u_i}$ are sometimes called the **spectrum** of **g**. This terminology comes from the Fourier basis set used in Fourier analysis. A discrete version of a Fourier basis set is similar to the Walsh set, except that the elements fit a sine wave pattern, and so are not binary-valued.

The orthonormal set of vectors we've defined above is said to be **complete**, because any vector in 8-space can be expressed as a linear weighted sum of these **basis vectors**. The weights are just the projections. If we had only 7 vectors in

our set, then we would not be able to express any 8-dimensional vector in terms of this basis set. The seven vector set would be said to be **incomplete**. A basis set which is orthonormal and complete is very nice from a mathematical point of view. Another bit of terminology is that these seven vectors would not **span** the 8-dimensional space. But they would span some sub-space, that is of smaller dimension, of the 8-space.

There has been much interest in describing the effective weighting properties of visual neurons in primary visual cortex of higher level mammals (cats, monkeys) in terms of basis vectors. One issue is if the input (e.g. an image) is projected (via a collection of receptive fields) onto a set of neurons, is information lost? If the set of weights representing the receptive fields of the collection of neurons is complete, then no information is lost.

■ Linear dependence

What if we had 9 vectors in our basis set used to represent vectors in 8-space? For the u's, it is easy to see that in a sense we have too many, because we could express the 9th in terms of a sum of the others. This set of nine vectors would be said to be linearly dependent. A set of vectors is linearly dependent if one or more of them can be expressed as a linear combination of some of the others. Sometimes there is an advantage to having an "over-complete" basis set (e.g. more than 8 vectors for 8-space; cf. Simoncelli et al., 1992).

Theorem: A set of mutually orthogonal vectors is linearly independent.

However, note it is quite possible to have a linearly independent set of vectors which are not orthogonal to each other. Imagine 3-space and 3 vectors which do not jointly lie on a plane. This set is linearly independent.

If we have a linearly independent set, say of 8 vectors for our 8-space, then no member can be dropped without a loss in the dimensionality of the space spanned.

It is useful to think about the meaning of linear independence in terms of geometry. A set of three linearly independent vectors can completely span 3-space. So any vector in 3-space can be represented as a weighted sum of these 3. If one of the members in our set of three can be expressed in terms of the other two, the set is not linearly independent and the set only spans a 2-dimensional subspace. That is, the set can only represent vectors which lay on a plane in 3-space. This can be easily seen to be true for the set of u's, but is also true for the set of v's.

■ Thought exercise

Suppose there are three inputs feeding into three neurons in a simple linear network. If the weight vectors of the three neurons are not linearly independent, do we lose information?

Basic matrix arithmetic

Definition of a matrix: a list of scalar lists

An mxn matrix has m rows, and n columns. Here is a 3x4 matrix:

In[73]:=
```
MatrixForm[
    Table[W[i,j],{i,1,3},{j,1,4}]]
```

Out[73]//MatrixForm=

$$\begin{pmatrix} W(1,1) & W(1,2) & W(1,3) & W(1,4) \\ W(2,1) & W(2,2) & W(2,3) & W(2,4) \\ W(3,1) & W(3,2) & W(3,3) & W(3,4) \end{pmatrix}$$

The matrix can be written in standard form using subscripts as:

$$\begin{pmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \end{pmatrix}$$

And as we have seen, Mathematica represents a matrix as a list of lists:

In[74]:=
```
FullForm[%]
```

Out[74]//FullForm=
```
{{W[1, 1], W[1, 2], W[1, 3], W[1, 4]},
 {W[2, 1], W[2, 2], W[2, 3], W[2, 4]}, {W[3, 1], W[3, 2], W[3, 3], W[3, 4]}}
```

Remember, the symbol % in Mathematica stands for the most recent output, %% stands for the second to last output, and so forth.

Adding, subtracting and multiplying by a scalar

As with vectors, matrices are added, subtracted, and multiplied by a scalar component by component.

In[88]:=
```
Clear[A,B,a,b,c,d,e,f,x,y,u,v,w,p,q,r,s,t]
A = {{a,b},{c,d}};
B = {{x,y},{u,v}}
```

Let's add A to B:

In[88]:=
```
A+B
```

Out[88]=
$$\begin{pmatrix} a+x & b+y \\ c+u & d+v \end{pmatrix}$$

Subtracting B from A:

In[89]:=
```
A-B
```

Out[89]=
$$\begin{pmatrix} a-x & b-y \\ c-u & d-v \end{pmatrix}$$

And if we multiply **A** by 3:

In[90]:=
```
3 A
```

Out[90]=
$$\begin{pmatrix} 3a & 3b \\ 3c & 3d \end{pmatrix}$$

Multiplying two matrices

We have already seen how to multiply a vector by a matrix: we replace the i^{th} row of the output vector by the inner product of the i^{th} row of the matrix with the vector.

In order to multiply a matrix **A**, by another matrix **B** to get **C = AB**, we calculate the ij^{th} component of the output matrix by taking the inner product of the i^{th} row of **A** with the j^{th} column of **B**:

In[91]:=
```
A.B
```

Out[91]=
$$\begin{pmatrix} bu+ax & bv+ay \\ du+cx & dv+cy \end{pmatrix}$$

Note that **AB** is not necessarily equal to **BA**:

In[92]:=
```
B.A
```

Out[92]=
$$\begin{pmatrix} ax+cy & bx+dy \\ au+cv & bu+dv \end{pmatrix}$$

Laws of commutation, association and distribution

Look at the element in the upper left of the matrix **BA** above--there is no reason, in general, for **ax+bu** to equal **ax + cy**. That is, matrix multiplication does not *commute*.

Apart from commutation for matrix multiplication, the usual laws of commutation, association, and distribution that hold for scalars hold for matrices. Matrix addition and subtraction do commute. Matrix multiplication is associative, so (**AB)C = A(BC)**. The distributive law works too:

A(B+C) = AB + AC

Non-square matrices

It is not necessary for **A** and **B** to be square matrices (i.e. have the same number of rows as columns) to multiply them. But if **A** is an mxn matrix, then **B** has to be an nxp matrix in order for **AB** to make sense. For example, here **F** is a 3x2 matrix, and **G** is a 2x4 matrix.

In[101]:=
```
F = {{a,b},{c,d},{e,f}};
G = {{p,q,r,s},{t,u,v,w}};
```

In[103]:=
```
Dimensions[F]
```

Out[103]= {3, 2}

In[104]:=
```
Dimensions[G]
```

Out[104]= {2, 4}

Because **F** has 2 columns, and **G** has 2 rows, it makes sense to multiply **G** by **F**:

In[105]:=
```
F.G
```

Out[105]=
$$\begin{pmatrix} a\,p+b\,t & a\,q+b\,u & a\,r+b\,v & a\,s+b\,w \\ c\,p+d\,t & c\,q+d\,u & c\,r+d\,v & c\,s+d\,w \\ e\,p+f\,t & e\,q+f\,u & e\,r+f\,v & e\,s+f\,w \end{pmatrix}$$

However, because the number of columns of **G** (4) do not match the number of rows of **F** (3), **G.F** is not well-defined:

In[106]:=
```
G.F
```

Dot::dotsh : Tensors $\begin{pmatrix} p & q & r & s \\ t & u & v & w \end{pmatrix}$ and $\begin{pmatrix} a & b \\ c & d \\ e & f \end{pmatrix}$ have incompatible shapes. ≫

Out[106]=
$$\begin{pmatrix} p & q & r & s \\ t & u & v & w \end{pmatrix} . \begin{pmatrix} a & b \\ c & d \\ e & f \end{pmatrix}$$

Inverse of a Matrix

Dividing a matrix by a matrix: the identity matrix & matrix inverses

The matrix corresponding to 1 or unity is the **identity matrix**. Like 1, the identity matrix is fundamental enough, that *Mathematica* provides a special function to generate n-dimensional identity matrices. Here is a 2x2:

In[107]:=
```
IdentityMatrix[2]
```

Out[107]=
$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

It is easy to show that the identity matrix plays the role for matrix arithmetic that the scalar 1 plays for scalar arithmetic.

How can one divide one matrix, say **B**, by another, say **A**? We can divide numbers, x by y, by multiplying x times the inverse of y, i.e. 1/y. So to do the equivalent of dividing **B** by **A**, we need to find a matrix **Q** such that when **A** is multiplied by **Q**, we get the matrix equivalent of unity, i.e. the identity matrix. Then "**B/A**" can be achieved by calculating the matrix product: **B.Q**.

In[108]:=
```
A = {{a,b},{c,d}};
```

Mathematica provides a built-in function to compute matrix inverses:

In[109]:=
```
Q = Inverse[A]
```

Out[109]=
$$\begin{pmatrix} \frac{d}{a\,d-b\,c} & -\frac{b}{a\,d-b\,c} \\ -\frac{c}{a\,d-b\,c} & \frac{a}{a\,d-b\,c} \end{pmatrix}$$

We can test to see whether the product of a A and Q is the identity matrix, but *Mathematica* won't go through the work of

simplifying the algebra in this case, unless we specifically ask it to.

In[110]:=
```
Q.A
```

Out[110]=
$$\begin{pmatrix} \frac{a\,d}{a\,d-b\,c}-\frac{b\,c}{a\,d-b\,c} & 0 \\ 0 & \frac{a\,d}{a\,d-b\,c}-\frac{b\,c}{a\,d-b\,c} \end{pmatrix}$$

In[111]:=
```
Simplify[Q.A]
```

Out[111]=
$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Here is a simple numerical example:

In[112]:=
```
B = {{1,-1},{3,2}};
R = Inverse[B]
```

Out[113]=
$$\begin{pmatrix} \frac{2}{5} & \frac{1}{5} \\ -\frac{3}{5} & \frac{1}{5} \end{pmatrix}$$

In[114]:=
```
B.R
```

Out[114]=
$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Badly conditioned matrices

What if one row is a scaled version of another row? Then the rows are not linearly independent. In this case, the inverse is not defined.

In[115]:=
```
B1 = {{1.5,1},{3,2.0}}
```

Out[115]=
$$\begin{pmatrix} 1.5 & 1 \\ 3 & 2. \end{pmatrix}$$

In[116]:=
```
Inverse[B1]
```

Inverse::sing : Matrix $\begin{pmatrix} 1.5 & 1. \\ 3. & 2. \end{pmatrix}$ is singular. ≫

Out[116]=
$$\begin{pmatrix} 1.5 & 1 \\ 3 & 2. \end{pmatrix}^{-1}$$

Sometimes the rows are almost, but not quite, linearly dependent (because the elements are represented as approximate floating point approximations to the actual values). *Mathematica* may warn you that the matrix is badly conditioned. Mathematica may try to find a solution to the inverse , but you should be suspicious of the solution. In general, one has to be careful of badly conditioned matrices.

Let's try finding the inverse、of the following matrix:

In[117]:=
```
B2 = {{-2,-1},{4.00000000000001,2.0}};
Inverse[B2]
```

Inverse::luc :

Result for Inverse of badly conditioned matrix $\begin{pmatrix} -2. & -1. \\ 4. & 2. \end{pmatrix}$ may contain significant numerical errors. ≫

Out[118]=
$$\begin{pmatrix} 2.04709 \times 10^{14} & 1.02355 \times 10^{14} \\ -4.09418 \times 10^{14} & -2.04709 \times 10^{14} \end{pmatrix}$$

In[119]:=
```
B2.Inverse[B2]
```

Inverse::luc :

Result for Inverse of badly conditioned matrix $\begin{pmatrix} -2. & -1. \\ 4. & 2. \end{pmatrix}$ may contain significant numerical errors. ≫

Out[119]=
$$\begin{pmatrix} 1. & 0. \\ 0. & 1. \end{pmatrix}$$

Were we lucky or not?

Determinant of a matrix

There is a scalar function of a matrix called the determinant. If a matrix has an inverse, then its determinant is non-zero. Does **B2** have an inverse?

In[120]:=
```
Det[B2]
```

Out[120]=
$$9.76996 \times 10^{-15}$$

Why do we get a zero determinant for **B2**, but not for **B1**?

In[121]:=
```
Det[B1]
```

Out[121]=
$$0.$$

Matrix transpose

We will use the transpose operation quite a bit in this course. It interchanges the rows and columns of a matrix:

In[122]:=
```
X = Table[w_{i,j}, {i, 1, 3}, {j, 1, 4}]
```

Out[122]=
$$\begin{pmatrix} w_{1,1} & w_{1,2} & w_{1,3} & w_{1,4} \\ w_{2,1} & w_{2,2} & w_{2,3} & w_{2,4} \\ w_{3,1} & w_{3,2} & w_{3,3} & w_{3,4} \end{pmatrix}$$

You may have noticed that in the Cell menu, you can convert to various cell types.

In StandardForm on the input line, the transpose is written:

In[123]:=
```
Transpose[X];
```

The output default is TradtionalForm. On the input line, in TraditionalForm, the transpose is written:

In[124]:=
```
X^T
```

Out[124]=
$$\begin{pmatrix} w_{1,1} & w_{2,1} & w_{3,1} \\ w_{1,2} & w_{2,2} & w_{3,2} \\ w_{1,3} & w_{2,3} & w_{3,3} \\ w_{1,4} & w_{2,4} & w_{3,4} \end{pmatrix}$$

What do the ouputs, **X** and X ^{T} look like in StandardForm, MatrixForm?

Getting parts of a matrix

We can pull out the i^{th} row of a matrix in *Mathematica* by simply writing W[[i]]. For example, the 2nd row of **X** is:

In[125]:=
```
X[[2]]
```

Out[125]=
$$\{w_{2,1}, w_{2,2}, w_{2,3}, w_{2,4}\}$$

What about i^{th} column of a matrix? There is no equally simple way of getting the column of a matrix in *Mathematica*, but we can use the transpose operation to do it. **Transpose[W] [[i]]** produces the ith column of matrix **W**. For example, the 3nd column of **X** is:

In[126]:=
```
Transpose[X][[3]]
```

Out[126]=
$$\{w_{1,3}, w_{2,3}, w_{3,3}\}$$

Symmetric matrices

For a square matrix, the diagonal elements remain the same under transpose.

If the transpose of a matrix equals itself, $\mathbf{H}^T = \mathbf{H}$, **H** is said to be a **symmetric matrix.** Symmetric matrices occur quite often in physical systems (e.g. the force on particle i by particle j is equal to the force of j on i). This means that the elements of a symmetric matrix **H** actually look like they are reflected about the diagonal.

In[127]:=
```
H = Table[N[Exp[-Abs[i-j]],1],{i,5},{j,5}]
```

Out[127]=
$$\begin{pmatrix} 1. & 0.4 & 0.1 & 0.05 & 0.02 \\ 0.4 & 1. & 0.4 & 0.1 & 0.05 \\ 0.1 & 0.4 & 1. & 0.4 & 0.1 \\ 0.05 & 0.1 & 0.4 & 1. & 0.4 \\ 0.02 & 0.05 & 0.1 & 0.4 & 1. \end{pmatrix}$$

■ Neural networks and symmetric connections

Do neural systems have symmetric connections? Real neural networks probably do not in general. Although, when the nature of the processing would not be expected to favor a particular asymmetry, we might expect that there should be symmetric connections on average. We made this assumption when setting up our lateral inhibition weight matrix, as seen above for **H**. Symmetric matrices have so many nice properties, that neural modelers (especially those from physics backgrounds) find the symmetry assumption almost irresistable. We'll see this later when we study the Hopfield networks. Lack of symmetry can have profound effects on the dynamics of non-linear networks and can produce chaotic trajectories of the state vector.

Outer product of two vectors

We've already seen that the inner product of two vectors produces a scalar. The **outer product** of an input and output vector can be used to model synaptic modification. It is can also be used calculate auto-covariances. Consider two vectors:

In[141]:=
```
Clear[f1,f2,f3,g1,g2,g3,f,g]
f = {f1,f2,f3};
g = {g1,g2,g3};
```

The outer product is just all of the pairwise products of the elements of **f** and **g** arranged in a nice (and special) order:

In[144]:=
```
h = Outer[Times,f,g]
```

Out[144]=
$$\begin{pmatrix} f1\,g1 & f1\,g2 & f1\,g3 \\ f2\,g1 & f2\,g2 & f2\,g3 \\ f3\,g1 & f3\,g2 & f3\,g3 \end{pmatrix}$$

The outer product is also written in traditional row and column format as: $\mathbf{f}\,\mathbf{g}^{\mathrm{T}}$

$$\mathbf{fg}^{T} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \end{pmatrix} \begin{pmatrix} g_1 g_2 g_3 \end{pmatrix}$$

Eigenvectors and eigenvalues

■ Eigenvectors

An eigenvector, **x**, of a matrix, **A**, is vector that when you multiply it by **A**, you get an output vector that points in the same direction as **x**:

$\mathbf{Ax} = \lambda\mathbf{x}$

where λ is a scalar that adjusts the length change of **x**.

There can't be any more than n distinct eigenvectors for an nxn matrix--and there may be less.

In[145]:=
```
A = {{1,2},{3,4}};
```

The *Mathematica* function **Eigenvectors[A]** returns the eigenvectors of matrix **A** as the rows of a matrix, which we'll call **eig**:

In[146]:=
```
eig = Eigenvectors[A]
```

Out[146]=
$$\begin{pmatrix} -\frac{4}{3}+\frac{1}{6}\left(5+\sqrt{33}\right) & 1 \\ -\frac{4}{3}+\frac{1}{6}\left(5-\sqrt{33}\right) & 1 \end{pmatrix}$$

We can verify that **eig[[1]]** and **A.eig[[1]]** lie along the same direction by taking the dot product of the unit vectors pointing in the directions of each:

In[150]:=
```
normalize[x_] := x/Sqrt[x.x];

normalize[eig[[1]]].normalize[A.eig[[1]]];
N[%]
```

Out[152]=
```
1.
```

■ Eigenvalues

The eigenvalues are given by:

In[153]:=
```
Eigenvalues[A]
```

Out[153]=
$$\left\{\frac{1}{2}\left(5+\sqrt{33}\right), \frac{1}{2}\left(5-\sqrt{33}\right)\right\}$$

Eigenvalues and eigenvector elements do not have to be real numbers. They can be complex, that is an element can be the sum of a real and imaginary number. In *Mathematica*, imaginary numbers are represented by multiples (or fractions) of **I**, the square root of -1:

In[154]:=
```
Sqrt[-1]
Sqrt[-1]//StandardForm
```

Out[154]=
$$i$$

Out[155]//StandardForm=
```
i
```

In[156]:=
```
B = {{1,2},{-3,4}};
Eigenvalues[B]
```

Out[157]=
$$\left\{\frac{1}{2}\left(5+i\sqrt{15}\right), \frac{1}{2}\left(5-i\sqrt{15}\right)\right\}$$

Linear systems analysis

■ Introduction

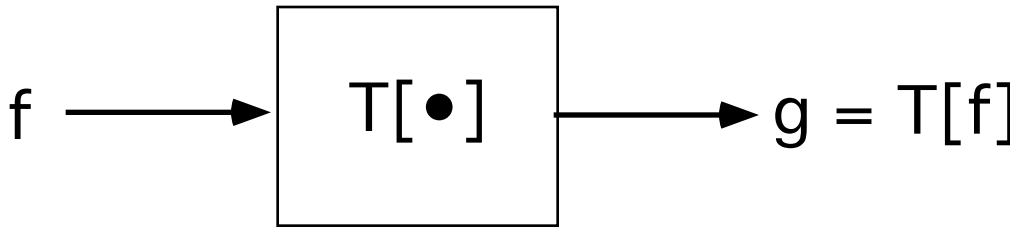
The world of input/output systems can be divided up into linear and non-linear systems. Linear systems are nice because the mathematics that describes them is not only well-known, but also has a mature elegance. On the other hand, it is a fair statement to say that most real-world systems are not linear, and thus hard to analyze...but fascinating if for that reason alone. Scientists were lucky with the limulus eye. That nature is usually non-linear doesn't mean one shouldn't familiarize oneself with the basics of linear system theory. Many times a non-linear system has a sufficiently smooth mapping that it can be approximated by a linear one over restricted ranges of parameter values.

So exactly what is a "linear system"?

The notion of a "linear system" is a generalization of the input/output properties of a straight line passing through zero. The matrix equation **W.x** == **y** is a linear system. This means that if **W** is a matrix, **x1** and **x2** are vectors, and a and b are scalars:

W.(a **x**1 + b **x2**) = a **W.x1** + b **W.x2**

This is a consequence of the laws of matrix algebra.The idea of a linear system has been generalized beyond matrix algebra. Imagine we have a box that takes inputs such as f, and outputs g = T[f].



The abstract definition of a linear system is that it satsifies:

$$T[a\ f + b\ g] = a\ T[f] + b\ T[g]$$

where T is the transformation that takes the sum of scaled inputs f, g (which can be functions or vectors) to the sum of the scaled transformation of f and g. The property, that the output of a sum is the sum of the outputs, is sometimes known as the *superposition principle* for linear systems. The fact that linear systems show superposition is good for doing theory, but as we will see later, it limits the kind of computations that can be done with linear systems, and thus with linear neural network models.

Characterizing a linear system by its response to an orthonormal basis set

Suppose we have an unknown physical system, which we model as a linear system **T**:

In[158]:=
```
T = Table[RandomReal[], {i, 1, 8}, {j, 1, 8}];
```

We would like to make a simple set of measurements that could characterize **T** in such a way that we could predict the output of **T** to any input. This is the sort of task that engineers face when wanting to characterize, say a stereo amplifier (as a model linear system), so that the output sound can be predicted for any input sound.What kind of measurements would tell us what **T** is? Well, we could just "stimulate" the system with cartesian vectors {1,0,0,0,0,0,0,0},{0,1,0,0,0,0,0,0}, and so forth and collect the responses which would be the columns of **T**. This has two practical problems: 1) for a real physical system, such as your stereo, or a neuron in the limulus eye, this would require stimulating it with a high-intensity audio or light intensity spike, which could damage what you are trying to study; 2) Characterizing the linear system by a matrix **T**, requires $\boldsymbol{n^2}$numbers, where **n** is the input signal vector length--and **n** can be pretty big for both audio and visual systems. Problem 2) has a nice solution when **T** is symmetric, and even nicer solution if the rows are shifted versions of each other (this is addressed later). Problem 1) can be addressed by showing that we can characterize **T** with any basis set--so we can pick one that won't blow out the physical system being tested.

The set of Walsh functions we looked at earlier is just one possible set that has the advantage that the elements that contribute to the "energy", i.e. (the square of the length) are distributed across the vector.

In[159]:=
```
Vectorlength[x_] := N[Sqrt[x.x]]
```

In[160]:=
```
v1 = {1, 1, 1, 1, 1, 1, 1, 1}; w1 = v1/Vectorlength[v1];
v2 = {1,-1,-1, 1, 1,-1,-1, 1}; w2 = v2/Vectorlength[v2];
v3 = {1, 1,-1,-1,-1,-1, 1, 1}; w3 = v3/Vectorlength[v3];
v4 = {1,-1, 1,-1,-1, 1,-1, 1}; w4 = v4/Vectorlength[v4];
v5 = {1, 1, 1, 1,-1,-1,-1,-1}; w5 = v5/Vectorlength[v5];
v6 = {1,-1,-1, 1,-1, 1, 1,-1}; w6 = v6/Vectorlength[v6];
v7 = {1, 1,-1,-1, 1, 1,-1,-1}; w7 = v7/Vectorlength[v7];
v8 = {1,-1, 1,-1, 1,-1, 1,-1}; w8 = v8/Vectorlength[v8];
```

We have already seen that the set $\{w_i\}$ spans 8-space in such a way that we can easily express any vector as a linear sum of these basis vectors.

$$g = \sum (g.w_i)\, w_i \qquad (1)$$

So as we saw before, an arbitrary vector, **g**

In[168]:=
```
g = {2,6,1,7,11,4,13, 29};
```

is the sum of its own projections onto the basis set:

In[169]:=
```
(g.w1) w1  + (g.w2) w2  +(g.w3) w3  +(g.w4) w4  +
(g.w5) w5  +(g.w6) w6  +(g.w7) w7  +(g.w8) w8
```

Out[169]=
```
{2., 6., 1., 7., 11., 4., 13., 29.}
```

Suppose we now do an "experiment" to find out how **T** transforms the vectors of our basis set:, and we put all of these transformed basis elements into a new set of vectors **newW[[i]]. newW** is a matrix for which each row is the response of **T** to a basis vector.

In[170]:=
```
newW = {T.w1,T.w2,T.w3,T.w4,T.w5,T.w6,T.w7,T.w8};
```

Note that **newW** is an 8x8 matrix. So how can we calculate the output of **T**, given **g** without actually running the input through **T**? If we do run the input through **T** we get:

In[171]:=
```
T.g
```

Out[171]=
```
{38.7249, 47.4927, 46.6253, 40.1691, 39.4675, 29.3432, 32.732, 30.6708}
```

But by the principle of linearity, we can also calculate the output by finding the "spectrum" of **g** as in Problem Set 2, and then scaling each of the transformed basis elements by the spectrum and adding them up:

$$T.g = T.\left\{\sum (g.w_i)\, w_i\right\} = \sum (g.w_i)\, T.w_i \qquad (2)$$

In[172]:=
```
(g.w1) T.w1 + (g.w2) T.w2 + (g.w3) T.w3 + (g.w4) T.w4 +
(g.w5) T.w5 + (g.w6) T.w6 + (g.w7) T.w7 + (g.w8) T.w8
```

Out[172]= {38.7249, 47.4927, 46.6253, 40.1691, 39.4675, 29.3432, 32.732, 30.6708}

Of course, we have already done our "experiment", so we know what the transformed basis vectors are, we stored them as rows of the matrix **newW**. We can calculate what the specturm (**g.w**i) is, so the output of **T** is:

In[173]:=
```
(g.w1) newW[[1]] + (g.w2) newW[[2]] + (g.w3) newW[[3]] +
(g.w4) newW[[4]] + (g.w5) newW[[5]] + (g.w6) newW[[6]] +
(g.w7) newW[[7]] + (g.w8) newW[[8]]
```

Out[173]= {38.7249, 47.4927, 46.6253, 40.1691, 39.4675, 29.3432, 32.732, 30.6708}

■ Same thing in more concise notation

Let the basis vectors be the rows of a matrix **W**:

In[174]:=
```
W = {w1, w2, w3, w4, w5, w6, w7, w8};
```

So again, we can project **g** onto the rows of **W**, and then reconstitute it in terms of **W** to get **g** back again:

In[175]:=
```
(W.g).W
```

Out[175]= {2., 6., 1., 7., 11., 4., 13., 29.}

In[176]:=
```
g.Transpose[W].newW
```

Out[176]= {38.7249, 47.4927, 46.6253, 40.1691, 39.4675, 29.3432, 32.732, 30.6708}

The main idea is: characterize and unknown system T by its response to orthonormal vectors. As an exercise, you can:

Show that T==Transpose[newW].W, and that the system output is thus: Transpose[newW].W.g

---

These new basis vectors do span 8-space, but they are not necessarily orthonormal. Under what conditions would they be orthogonal? What if the matrix T was symmetric, and we deliberately chose the basis set to describe our input to be the eigenvectors of T?

What if the choice of basis set is the set of eigenvectors of T?

We've seen how linearity provides us with a method for characterizing a linear system in terms of the responses of the system to the basis vectors. The problem is that if the input signals are long vectors, say with dimension 40,000, then this set of basis vector responses is really big--1.6×10^9.

Construct a symmetric matrix transformation, T. Show that if the elements of the basis set are the eigenvectors of T, then the transformation of any arbitrary input vector x is given by:

$$T[\mathbf{x}] = \sum \alpha_i \lambda_i \mathbf{e}_i$$

Where the α_i are the projections of x onto each eigenvector. Having the eigenvectors of **T** enables us to express the input and output of **T** in terms of the same basis set--the eigenvectors. All **T** does to the input is to scale its projection onto each eigenvector by the eigenvalue for that eigenvector. The set of these eigenvalues is sometimes called the *modulation transfer function* because it describes how the amplitude of the eigenvectors change as they pass through **T**.

Linear systems analysis is the foundation of Fourier analysis, and is why it makes sense to characterize your stereo amplifier in terms of frequency response. But your stereo isn't just any linear system--it has the special property that it can input a sound at time t and measure the response, and then you input the same sound again at a later time, you get the same response, except of course that is is shifted in time. It is said to be a shift-invariant system. The eigenvectors of a shift-invariant system are sinusoids. (The eigenvectors of the symmetric matrix are sinusoids, not just because the matrix was symmetric, but also because each row of the matrix was a shifted version of the previous row--the elements along any given diagonal are identical. This is called a symmetric Toeplitz matrix.)

Sinewave inputs are the eigenvectors of your stereo system. The dimensionality is much higher--if you are interested in frequencies up to 20,000 Hz, your eigenvector for this highest frequency would have least 40,000 elements--not just 8!

This kind of analysis has been applied not only to physical systems, but to a wide range of neural sensory systems. For the visual system alone, linear systems analysis has been used to study the cat retina (Enroth-Cugell and Kopson, 1966), the monkey visual cortex, and the human contrast sensitivity system as a whole (Campbell and Kopson, 1968).

Much empirical analysis has been done using linear systems theory to characterize neural sensory systems, and other neural systems such as those for eye movements. It works surprisingly well as long as the linear system approximation holds.

And it does do quite well for the lateral eye of the limulus, X-cells and P-cells of the mammalian visual system, over restricted ranges for so-called "simple" cells in the visual cortex, among others. The optics of the simple eye is another example of an approximately linear system. Many non-linear systems can be approximated as linear systems over smooth subdomains.

In summary, if T has n distinct orthogonal eigenvectors, $\mathbf{e}_i$, and known eigenvalues, λ_i, then:

Step 1: Project **x** onto eigenvectors of T: $\mathbf{x}.\mathbf{e}_i$

Step 2: Scale each $\mathbf{x}.\mathbf{e}_i$ by the eigenvalue of $\mathbf{e}_i$: $\lambda_i \mathbf{x}.\mathbf{e}_i$

Step 3: Scale each $\mathbf{e}_i$ by $\lambda_i \mathbf{x}.\mathbf{e}_i$

Step 4: Sum these up. That's the response of T to **x**! : $\sum_i \lambda_i \boldsymbol{x}.\boldsymbol{e}_i$

Optional: eigenvectors, matrices and solving algebraic equations

Eigenvectors, eigenvalues: algebraic manipulation

Last time we used the *Mathematica* function **Eigenvectors[]** and **Eigenvalues[]** to produce the eigenvectors and eigenvalues for the matrix equation: **Ax = λx**. It is worth spending a little time to understand what is being done algebraically. Consider the following pair of equations specified by a 2x2 matrix **W** acting on **xv** to produce a scaled version of **xv**:

In[177]:=
```
W = {{1,2},{2,1}};
xv := {x,y};
lambda xv == W.xv
```

Out[179]=
$$\{\text{lambda}\, x, \text{lambda}\, y\} = \{x + 2\, y, 2\, x + y\}$$

Finding the eigenvalues is a problem in solving this set of equations. If we eliminate x and y from the pair of equations, we end up with a quadratic equation in lambda:

■ Eliminate[] & Solve[]

In[180]:=
```
Eliminate[{lambda xv == W.xv, lambda != 0}, {x, y}]
```

Out[180]=
$$(\text{lambda} - 3 \neq 0 \bigwedge \text{lambda} \neq 0 \bigwedge \text{lambda} + 1 \neq 0) \bigvee \text{lambda}^2 - 2\, \text{lambda} = 3$$

In[181]:=
```
Solve[-2 lambda + lambda^2 == 3, lambda]
```

Out[181]=
$$\{\{\text{lambda} \to -1\}, \{\text{lambda} \to 3\}\}$$

So our eigenvalues are -1 and 3. We can plug these values of lambda into our equations to solve for the eigenvectors:

In[182]:=
```
Solve[{-x == x + 2 y, - y == 2 x + y}, {x,y}]
Solve[{3 x == x + 2 y, 3 y == 2 x + y}, {x,y}]
```

Solve::svars : Equations may not give solutions for all "solve" variables. ≫

Out[182]=
$$\{\{x \to -y\}\}$$

Solve::svars : Equations may not give solutions for all "solve" variables. ≫

Out[183]=
$$\{\{x \to y\}\}$$

■ Reduce

Mathematica is smart enough that we can use **Reduce[]** to do it all in one line:

In[184]:=
```
Reduce[{lambda xv == W.xv}, {x,y,lambda}]
```

Out[184]=
$$(x = 0 \land y = 0) \bigvee \left((y = -x \lor y = x) \bigwedge x \neq 0 \bigwedge \text{lambda} = \frac{x + 2\,y}{x} \right)$$

The eigenvectors are unique only up to a scale factor, so one can choose how to normalize them. For example, we could arbitrarily set x to 1, and then the eigenvectors are: {1,1}, and {1,-1}. Alternatively, we could normalize them to {1/-Sqrt[2], 1/Sqrt[2]} and {1/Sqrt[2], -1/Sqrt[2]}.

In[185]:=
```
Eigenvectors[W]
```

Out[185]=
$$\begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix}$$

■ Side note: Solve[] vs. Reduce[]

Solve[] makes assumptions about constraints left unspecified, so the following returns the solution true for any lambda:

In[186]:=
```
Solve[{lambda x == x + 2 y, lambda y == 2 x + y},
    {x,y,lambda}]
```

Solve::svars : Equations may not give solutions for all "solve" variables. ≫

Out[186]= $\{\{x \to -y, \text{lambda} \to -1\}, \{x \to y, \text{lambda} \to 3\}, \{x \to 0, y \to 0\}\}$

In[187]:=
```
Solve[{lambda x == x + 2 y, lambda y == 2 x + y,
lambda !=0, x != 0, y != 0}, {x,y,lambda}]
```

Solve::svars : Equations may not give solutions for all "solve" variables. ≫

Out[187]= $\{\{x \to -y, \text{lambda} \to -1\}, \{x \to y, \text{lambda} \to 3\}\}$

Reduce[] gives all the possibilities without making specific assumptions about the parameters:

Either of the following forms will work too:

In[188]:=
```
Reduce[lambda xv == W.xv, {xv[[1]],xv[[2]],lambda}]
Reduce[{{lambda x, lambda y} == {x + 2 y, 2 x + y}},
    {x,y,lambda}]
```

Out[188]= $(x = 0 \land y = 0) \bigvee \left((y = -x \bigvee y = x) \bigwedge x \neq 0 \bigwedge \text{lambda} = \frac{x + 2\,y}{x} \right)$

Out[189]= $(x = 0 \land y = 0) \bigvee \left((y = -x \bigvee y = x) \bigwedge x \neq 0 \bigwedge \text{lambda} = \frac{x + 2\,y}{x} \right)$

■ Determinant solution

Ax = λx can be written: **(A - Iλ).x, w**here **I** is the identity matrix. The interesting values of **x** that satisfy this equation are the ones that aren't zero. For this to be true, **(A - Iλ)** must be singular (i.e. no inverse). And this is true if the determinant of **(A - Iλ)** is zero

References

Simoncelli, E. P., Freeman, W. T., Adelson, E. H., & Heeger, D. J. (1992). Shiftable Multi-scale Transforms. IEEE Trans. Information Theory, 38(2), 587--607.

Strang, G. (1988). Linear Algebra and Its Applications (3rd ed.). Saunders College Publishing Harcourt Brace Jovanovich College Publishers.

See also: http://ocw.mit.edu/OcwWeb/Mathematics/18-06Spring-2005/CourseHome/