Matrix math for data analysis: left divide and `lsqnonneg`

Andrew B. Greytak[*]
Department of Chemistry and Biochemistry, University of South Carolina, Columbia, SC 29208

June 19, 2024

1 Arrays in Matlab

Be sure to read "Basic Matrix Operations" section of Matlab documentation! Matlab was originally designed to handle matrix math: multiplication, division, determinants, eigenvalues, etc and those tasks can be accomplished very easily and concisely. The Matlab language, like other computer languages, is able to have variables that store data in arrays of zero (scalar), one (vector), two (typical matrix), or more dimensions. However, the language has a couple of features that are legacies of its use for array math:

1. indexing starts from 1
2. even for vectors, it keeps in mind whether it is a row vector or a column vector, and many operations will work differently or not at all if the array is the wrong "shape". You can easily transpose a matrix with `'`.
3. in a 2 dimensional array, the first index is always row and the second is always column: `a(row,col)`
4. Matlab always assumes that `*`, `/`, and exponents are matrix operations unless you insist on element-wise operations with a leading `.`

```
>> a = [ 1 5 10 ; 16 27 2 ]
a =
     1     5    10
    16    27     2
>> a(2,1)
ans =
    16
>> size(a)
ans =
     2     3
```

Note that, as with other languages, you can also address items in an array with a single "linear" address. When Matlab does this, it reads down each column in order. This is called **column-major** behavior:

```
>> a(5)
ans = 10
```

All items in an array must be of the same data type (integer, single or double precision floating point [default], structures with the same field names in same order). Most "real" matrix math is only defined for floating point data types, but you can multiply arrays of integers by a scalar.

[*]email: greytak@sc.edu

2 Systems of linear equations

There are two situations we frequently encounter in our lab that can be quickly solved using matrix division, because they can be though of as systems of linear equations. One is performing a linear or polynomial curve fit to describe how one value depends on another. The other is solving a linear combination, where we try to express one function as a sum of two or more known components defined over the same space (such as using the absorption spectrum of a mixture of two dyes to determine the concentrations of each).

y vs x data described by a linear function This is a classic linear fit. You imagine there is some function $y = ax + b$ that describes your data. If you have just 2 data points (x_1, y_1) and (x_2, y_2), you would have two equations and two unknowns: the coefficient a and constant b:

$$y_1 = ax_1 + b \tag{1}$$

$$y_2 = ax_2 + b \tag{2}$$

You can write this as a matrix equation:

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} x_1 & 1 \\ x_2 & 1 \end{bmatrix} \times \begin{bmatrix} a \\ b \end{bmatrix} \tag{3}$$

or

$$\mathbf{y} = \mathbf{M} \times \mathbf{A} \tag{4}$$

where $\mathbf{y}$ is a column vector of known y values, $\mathbf{M}$ is a matrix containing known x values padded with a column of 1's, and $\mathbf{A}$ is a column vector of coefficients. If there are just 2 data points, it'll always be possible to find an exact solution – a vector (pair in this case) of coefficients that solves equation 4. With 2 equations and 2 unknowns, $\mathbf{M}$ is a square matrix and can be inverted. So, $\mathbf{A}$ can be found as

$$\mathbf{A} = \mathbf{M}^{-1} \times \mathbf{M} \times \mathbf{A} = \mathbf{M}^{-1} \times \mathbf{y} \tag{5}$$

This operation – multiplication on the left by the inverse of a matrix – is known in Matlab as "left division" and can be entered concisely as a backslash.

```
A = M\y
```

The behavior of Matlab when you type this is to find A such that

$$\mathbf{M} \times \mathbf{A} - \mathbf{y} = \mathbf{0} \tag{6}$$

Once you know the coefficients, you can use them to calculate y for any number of x values:

$$\begin{bmatrix} y_1 \\ y_2 \\ \ldots \\ y_n \end{bmatrix} = \begin{bmatrix} x_1 & 1 \\ x_2 & 1 \\ \ldots & \ldots \\ x_n & 1 \end{bmatrix} \times \begin{bmatrix} a \\ b \end{bmatrix} \tag{7}$$

This matrix multiplication is perfectly acceptable, but if someone gives you more than 2 pairs of x and y values, you cannot be sure there is an exact solution for $\mathbf{A}$. Two points always define a line, three points might not. The problem is **overdetermined**. Luckily, Matlab deals seamlessly with overdetermined matrix division: it simply gives you the "least-squares solution": the value of $\mathbf{A}$ that minimizes the difference (specifically the norm or sum of squared differences) between $\mathbf{y}$ and $\mathbf{M} \times \mathbf{A}$. If you type

```
A = M\y
```

it will give:

$$\min_{\mathbf{A}} ||\mathbf{M} \times \mathbf{A} - \mathbf{y}||^2 \tag{8}$$

2.1 Linear fit example

Let's try it out. Exact case:

```
>> y=[7 ; 11]
y =
     7
    11
>> x=[2 ; 4]
x =
     2
     4
>> M=[x ones(size(x))]
M =
     2     1
     4     1
>> A=M\y
A =
     2
     3
```

So we see that $a = 2$, $b = 3$, and $y = 2x + 3$ describes all of our data.

Linear fit (overdetermined case):

```
>> y=[7 ; 11 ; 16]
y =
     7
    11
    16
>> x=[2 ; 4 ; 6]
x =
     2
     4
     6
>> M=[x ones(size(x))]     % could also get the column of 1's by M=[x x.^0]
M =
     2     1
     4     1
     6     1
>> A=M\y
A =
    2.2500
    2.3333
>> y_fit=M*A
y_fit =
    6.8333
   11.3333
   15.8333
```

So we see that the least-squares linear fit is $y = 2.25x + 2.3333$.

2.2 Polynomial fit

The same approach can be extended to polynomial fits (y as a quadratic, cubic, etc function of x). The polynomial fitting is still a linear algebra problem because you already know the x values, so it is easy to calculate a column equal to x^2 or x^3 and add it to our matrix. The vector of coefficients, A, simply expands to include a row for each coefficient including the constant term. A function $y = ax^2 + bx + c$ can be calculated for every value of x as:

$$\begin{bmatrix} y_1 \\ y_2 \\ \cdots \\ y_n \end{bmatrix} = \begin{bmatrix} x_1^2\ x_1\ 1 \\ x_2^2\ x_2\ 1 \\ \cdots\ \cdots \\ x_n^2\ x_n\ 1 \end{bmatrix} \times \begin{bmatrix} a \\ b \\ c \end{bmatrix} \tag{9}$$

$$\mathbf{y} = \mathbf{M} \times \mathbf{A} \tag{10}$$

If we are given 3 or more pairs of x and y values, we can obtain the coefficients for a least-squares quadratic fit as:

```
A = M\y
```

As an example:

```
>> y=[3; 6; 11; 18]
y =
     3
     6
    11
    18
>> x=[1; 2; 3; 4]
x =
     1
     2
     3
     4
>> M=[x.^2 x ones(size(x))]     % or M=[x.^2 x.^1 x.^0]
M =

     1     1     1
     4     2     1
     9     3     1
    16     4     1
>> A=M\y
A =
    1.0000
   -0.0000
    2.0000
```

So we see that the least-squares quadratic fit (which happens to be perfect in this case) has $a = 1$, $b = 0$, $c = 2$, so $y = x^2 + 2$. Note that Matlab has many tools for dealing with polynomials, including `poly` which gets the coefficients of a polynomial given its roots (a relatively easy calculation) and `roots` which gets the roots of a polynomial given the coefficients (such as solving a quadratic or cubic equation), a much harder problem that must be (and is) accomplished numerically for higher orders.

Note that while people often reach for polynomial fits to describe trends in data, especially when the proper equation to describe the underlying principle is unknown, there are two caveats. Firstly, if the underlying principles do not follow a polynomial response, the parameters you get will not correspond to any fundamental characteristics of the system. Secondly, polynomial fits tend to "explode" to unphysical values outside of the range of data they were trained on, so in any application in which you use a function obtained from a polynomial fit, you need to know over what range of x values is valid (don't use values outside of the fit range).

3 Linear combinations

3.1 General approach

Here we assume that we are interested in y values not as a function of x, but as a function of some index (that perhaps describes a wavelength: one absorbance value for each of a number of wavelengths, and we have the wavelength value for each value of the index somewhere, but it isn't immediately important to our discussion). The key thing is we also have have the values of two (or more) other spectra at the same set of wavelengths, and we think our y is a linear combination of these two (or more) components. Every value of y is obtained by multiplying the each component spectrum by a constant coefficient, and adding them up.

$$y_1 = aw_{1,1} + bw_{1,2} \tag{11}$$

$$y_2 = aw_{2,1} + bw_{2,2} \tag{12}$$

Here, the $w_{j,k}$ are the values of component spectrum k at (wavelength) index j, and coefficients a and b are the amplitudes (weighting factors or concentrations) for components 1 and 2 respectively. Once again we can write this as a matrix equation:

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} w_{1,1} \; w_{1,2} \\ w_{2,1} \; w_{2,2} \end{bmatrix} \times \begin{bmatrix} a \\ b \end{bmatrix} \tag{13}$$

or

$$\mathbf{y} = \mathbf{M} \times \mathbf{A} \tag{14}$$

with

$$\mathbf{M} = \begin{bmatrix} w_{1,1} \; w_{1,2} \\ w_{2,1} \; w_{2,2} \end{bmatrix} = \begin{bmatrix} \mathbf{w_1} \; \mathbf{w_2} \end{bmatrix} \tag{15}$$

where $\mathbf{w_1}$ and $\mathbf{w_2}$ are column vectors containing the values of each component spectrum at each index of column vector $\mathbf{y}$. If we are given 2 component spectra, and are given 2 y values, we can always find a and b as

```
A = M\y
```

The unknown components of $\mathbf{A}$, a and b, could be taken as concentrations of two components, or used to describe the *total* intensity and *ratio* of intensities for a ratiometric fluorescent sensor, for example. However, we often want to consider more y values than components: for example, you have a mixture of two components but record an absorption spectrum that includes absorbance values at many wavelengths. As before, in this case the problem will be overdeterimined, but Matlab can quickly give you a least-squares solution:

$$\begin{bmatrix} y_1 \\ y_2 \\ \ldots \\ y_n \end{bmatrix} = \begin{bmatrix} w_{1,1} \; w_{1,2} \\ w_{2,1} \; w_{2,2} \\ \ldots \; \ldots \\ w_{n,1} \; w_{n,2} \end{bmatrix} \times \begin{bmatrix} a \\ b \end{bmatrix} \tag{16}$$

and

```
A = M\y
```

will give:

$$\min_{\mathbf{A}} ||\mathbf{M} \times \mathbf{A} - \mathbf{y}||^2 \tag{17}$$

The best-fit linear combination can then be obtained using the coefficients $\mathbf{A}$:

$$\mathbf{y_{fit}} = \mathbf{M} \times \mathbf{A} \tag{18}$$

Evaluating the residual error by comparing this fit to the original data, which is only possible if you have more data points than components, is very helpful in deciding whether or not your data is actually described as a linear combination of these components, or if there is some other contribution that you haven't accounted for. A flat background can be considered by introducing a constant value as one of the components.

3.2 Avoiding negative coefficients: `lsqnonneg`

When Matlab does the "left divide" it does not presume anything about the values of the coefficients and they could be negative. In many practical situations, your data (such as an absorption spectrum) may well be approximated by a linear combination of several components but the coefficients in the linear combination (such as concentrations of the components in the mixture) cannot be negative. Luckily, Matlab has a built-in function that can deal with this called `lsqnonneg`. The order of arguments is the same: instead of

```
A = M\y
```

you would write

```
A = lsqnonneg(M,y)
```

which solves

$$\min_{\mathbf{A}\geq\mathbf{0}} ||\mathbf{M}\times\mathbf{A}-\mathbf{y}||^2 \qquad (19)$$

where the requirement is that every element of $\mathbf{A}$ is not less than zero. Once again, once you have coefficients $\mathbf{A}$, the linear combination fit can be obtained as

$$\mathbf{y_{fit}} = \mathbf{M}\times\mathbf{A} \qquad (20)$$

Above, we have presumed that we know what components $\mathbf{w_k}$ might be present. In some cases, it might be possible to vary the form of the component vectors in an effort to minimize the residual error when the linear combination fit is applied. For example, you might be able to find the spectrum of the acid and base forms of a pH-sensitive dye, and the pKa, if you have absorption spectra at a range of known pH's, since you know how the amplitude (concentration) of each component should vary with pH. Our `getpKa` function can do this.

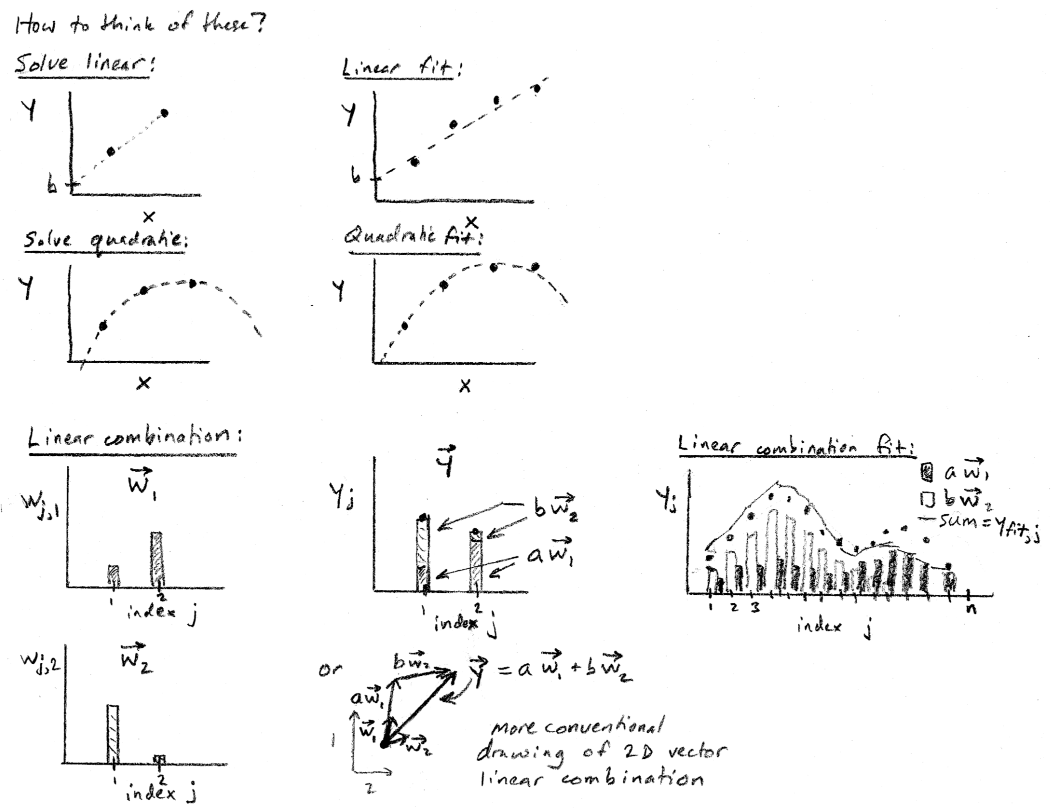

Figure 1: Visualizing some of the approaches described here.