summary on Least-squares

A linear system with more equations than unknowns is said to be **overdetermined**. Usually, such a system has no solution. What to do? That depends on why you are considering the linear system in the first place. If we think of the linear system

$$A? = b$$

as the attempt to write b as a weighted sum of the columns of A, then it makes sense to look for x for which the residual, $b - Ax$, is as small as possible. This is most easily done if we measure the size of $b - Ax$ by its Euclidean norm

$$\|b - Ax\|_2 = \sqrt{(b_1 - (Ax)_1)^2 + (b_2 - (Ax)_2)^2 + \cdots},$$

and minimizing it is the same as minimizing

$$\|b - Ax\|_2^2 = (b_1 - (Ax)_1)^2 + (b_2 - (Ax)_2)^2 + \cdots.$$

For this reason, the x that makes this sum as small as possible is called the **least-squares solution** of $A? = b$.

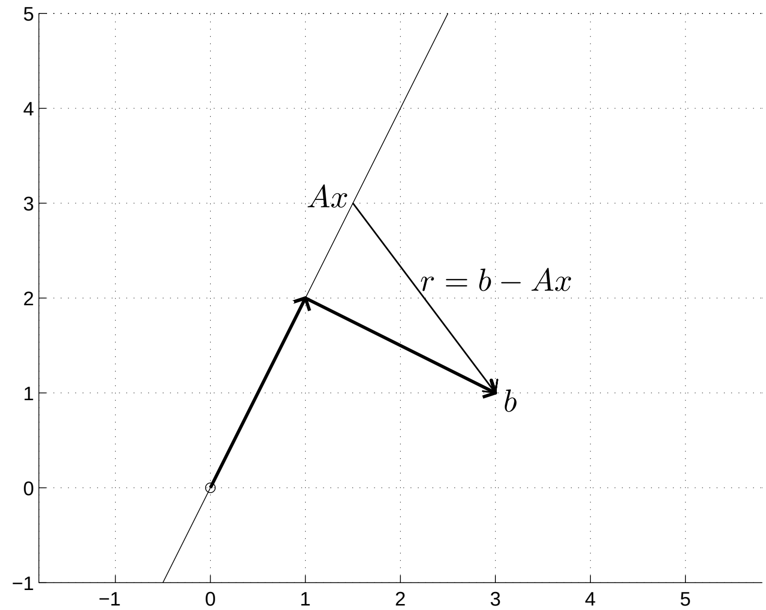
The simplest example occurs when A has just one column but two rows, e.g.,

$$A := \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \quad b := \begin{bmatrix} 3 \\ 1 \end{bmatrix}.$$

Now we are seeking x_1 so that the point $Ax = (x_1, 2x_1)$ in the plane is as close as possible to the point $b = (3, 1)$. The set of all points of the form $(x_1, 2x_1)$ is a straight line, the line **spanned by** $(1, 2)$, and the point Ax on that line closest to b (in the Euclidean norm) is characterized by the fact that its residual is perpendicular to that line, i.e., $A'(b - Ax) = 0$, or

(1) $$A'Ax = A'b.$$

For our simple example, $A'A = [5]$, $A'b = [5]$, hence $x_1 = 1$, as is shown in the figure:



This simple example illustrates the fact that, in general, the least-squares solution for $A? = b$ satisfies the **normal equations** (1), hence is uniquely determined provided $A'A$ is invertible. (One can show that $A'A$ is invertible if and only if A is 1-1, i.e., if and only if the only way we can write the zero vector as a weighted sum Ax of the columns of A is the trivial way, i.e., with x the zero-vector.) Note that the 'normal' equations derive their name from the fact that they express the requirement that the residual be 'normal', i.e., perpendicular, to the columns of A.

One used to determine the least-squares solution by solving the normal equations. However, it was realized that there is a stabler way for determining the least-squares solution, namely via the QR factorization for A. This factorization, available in `matlab` via `[Q,R] = qr(A)`, provides a unitary matrix Q and a right-triangular or upper-triangular matrix R for which $A = QR$. Recall that a matrix Q is **unitary** if

$$Q^{-1} = Q'.$$

Further, R is **right-triangular** if it is upper triangular, i.e., if $R(i,j) = 0$ for all $i > j$, i.e., all entries of R strictly below the main diagonal are zero.

Unitary matrices are important here because they preserve the Euclidean norm:

$$\|Qx\|_2 = \|x\|_2$$

for all x, as can be seen at once from the calculation

$$\|Qx\|_2^2 = (Qx)'(Qx) = x'Q'Qx = x'x = \|x\|_2^2.$$

So, if $A = QR$ with Q unitary, then also Q' is unitary, hence

$$\|b - Ax\|_2 = \|Q'(b - Ax)\|_2 = \|Q'b - Rx\|_2.$$

Now, let's be precise about sizes; assume that A is $n \times k$. Then also R is $n \times k$. Since R is right-triangular, $(Rx)(k+1{:}n)$ is zero regardless of x. Hence, in trying to make $\|Q'b - Rx\|_2$ small by proper choice of x, we can't do anything about $(Q'b - Rx)(k+1{:}n) = (Q'b)(k+1{:}n)$. The best we can do is make $(Q'b - Rx)(1{:}k)$ equal to zero, assuming that $R1 := R(1{:}k, :)$ is invertible (as it will have to be in case A is 1-1). In that case, $R1$ is upper triangular, hence we can solve the system

$$R1x = (Q'b)(1{:}k)$$

by back-substitution. Note that we don't need all of Q for this; since

$$(Q'b)(1{:}k) = Q'(1{:}k, :)b = Q(:, 1{:}k)'b,$$

we only need $Q1 := Q(:, 1{:}k)$, i.e., the first k columns of Q.

The `matlab` command `[Q1,R1] = qr(A,0);` explicitly provides $R1$ as well as $Q1$, hence the two commands

```
[Q1,R1] = qr(A,0); x = R1\(Q1'*b);
```

supply the least-squares solution to $A? = b$. Better yet, the single command

```
x = A\b;
```

does the same thing (using, in effect, the QR factorization for A).

Obtaining the least-squares solution this way is to be preferred to solving the normal equations for it because the condition number of $R1$ is the squareroot of the condition number of $A'A$:

$$\kappa(A'A) = \kappa(R1)^2 = \kappa(A)^2.$$

Least-squares data fitting

Overdetermined linear systems appear routinely in data fitting: One is given data x_i, y_i, $i = 1{:}n$, but does not want to interpolate, perhaps because the data are noisy and/or because one wants do *reduce the data*, i.e., fit the data with a model that uses fewer than n degrees of freedom.

A standard example is the straight line fit in which one determines the coefficients $c1$ and $c2$ so as to minimize

$$\sum_{i=1}^{n} (y_i - (c1\, x_i + c2))^2.$$

This makes $c = (c1, c2)$ the least-squares solution to the linear system

$$x_i\, c1 + c2 = y_i, \quad i = 1{:}n.$$

Hence, assuming that the column vectors x and y contain the data, we get the solution in matlab via

```
c = [x ones(size(x))]\y;
```

We could fit such data by higher-degree polynomials, e.g., by a cubic polynomial, in which case the best coefficients are obtained by

```
c = [(x-m).^3 (x-m).^2 x-m ones(size(x))]\y;
```

where m is chosen so as to reduce the condition number of this Vandermonde matrix; e.g., m=mean(x). Actually, matlab's command

```
c = polyfit(x-m,y,3);
```

accomplishes the same thing (without your having to generate that Vandermonde matrix explicitly), and c=polyfit(x-m,y,1) would have provided the straight-line least-squares fit. matlab's polyval(c,z-m) then supplies the value(s) at z of the resulting cubic polynomial fit.

The very same idea applies to the least-squares fitting of any kind of *model*

$$y \approx c_1\varphi_1(x) + c_2\varphi_2(x) + \cdots + c_k\varphi_k(x)$$

to the data. Now we are looking for the least-squares solution of the linear system

$$c_1\varphi_1(x_i) + c_2\varphi_2(x_i) + \cdots + c_k\varphi_k(x_i) = y_i, \quad i = 1{:}n.$$

Hence, in matlab, the best choice of the coefficient vector $c = (c_1, \ldots, c_k)$ can be computed by

```
A = zeros(n,k);
for j=1:k
   A(:,j) = phi(j,x);
end
c = A\y;
```

This assumes that we have a function values = phi(j,x) that returns the value(s) at x of φ_j.

OPTIONAL example, not required reading!!

As a final, amusing example, suppose that we want to fit the data by a cubic spline, with breakpoints $\xi_1 < \xi_2 < \cdots < \xi_k$. We know that `matlab`'s command `spline(xi,eta,x)` will return the values at `x` of the cubic spline interpolant with the not-a-knot end condition that interpolates the value `eta(j)` at its breakpoint `xi(j)`, `j=1:k`. In particular, the commands

```
Ik = eye(k);
A = zeros(n,k);
for j=1:k
   A(:,j) = spline(xi,Ik(:,j),x);
end
```

will generate the matrix `A` whose jth column contains the values at `x` of the not-a-knot cubic spline L_j that is zero at `xi(m)` for all `m` not equal to `j`, and is equal to 1 at `xi(j)`. This should remind you of the Lagrange polynomials we used during the discussion of quadrature rules. In particular, the not-a-knot spline interpolant provided by `spline(xi,eta)` can be written

$$\eta_1 L_1 + \eta_2 L_2 + \cdots + \eta_k L_k. \tag{2}$$

This gives us an explicit model for the not-a-knot cubic spline with breakpoints $\xi_1 < \cdots < \xi_k$, and we can use it to determine the least-squares spline fit by this model to given data `x`, `y`. With the matrix `A` as generated in the preceding fragment, we get

```
eta = A\y;
```

for the best choice of the coefficients $(\eta_1, \ldots, \eta_k)$ in our spline model (2). To get the actual least-squares cubic spline from it, we get it as the not-a-knot spline interpolant to the value η_j at ξ_j, $j = 1{:}k$, i.e., as

```
l2cs = spline(xi,eta);
```

Now, actually, the version of `spline` in the m-files subdirectory for CS412 can even handle *vector-valued* functions. This means that this entire calculation can be done in just one statement

```
l2cs = spline(xi,spline(xi,eye(k),x').'\y);
```

but this may well be too hard to understand at first reading. To explain: If `y` is $d \times k$, and `x` is $n \times 1$, then the statement `spline(xi,y,x')` returns a matrix of size $d \times n$, with its ith column the 'value' at `x(i)` of the not-a-knot cubic spline that matches the 'value' `y(:,j)` at `x(j)`, `j=1:n`. In other word, the jth *row* of `spline(xi, y, x')` contains exactly the result of `spline(xi,y(j,:),x')`. This guarantees that the jth row of `spline(xi,eye(k),x')` contains the values at `x` of the 'Lagrange spline' L_j, `j=1:k`, and so explains why its transpose is used instead.

Finally, the cubic splines used here all satisfy the not-a-knot condition, i.e., the first and last interior breakpoint isn't actually a breakpoint. On the other hand, if `eta` has two more entries than `xi`, then `spline(xi,eta,x)` provides the values at `x` of the complete (or clamped) cubic spline interpolant. Correspondingly, the statement

```
l2cs = spline(xi,spline(xi,eye(k+2),x').'\y);
```

provides the least-squares approximation from the set of all cubic splines with breakpoint sequence `xi`.