Reading them in

Here is the script that reads in the proteins, showing their order:

```matlab
% script to load the 7 proteins for project 2. Result is a % 7-element cell-array called "proteins"

names = {'1MBD.pdb' '1SDK.pdb' '6HBI.pdb' '4VHB.pdb' ...
         '2GDM.pdb' '1MBC.pdb' '102L.pdb'};

for i = 1:length(names)
    proteins{i} = pickCA(names{i});
end

proteins{2} = proteins{2}{1}; % A chain
proteins{3} = proteins{3}{1}; % A chain
proteins{4} = proteins{4}{2}; % B chain
```

We will use the word “protein” to refer to one of these seven chains, and we’ll use the word “structure” to refer to any sequence of C-alpha atoms. Unless stated otherwise, when speaking of two structures, we mean two structures of equal length.

The project consists of several parts. The first part gives code that minimizes the distance between two structures (of equal length—this won’t be mentioned again) by optimally translating one with respect to the other. The second part gives code that minimizes the distance between two structures by optimally rotating one with respect to the other. The third part uses the first two parts to compute the distances between the seven proteins. The fourth part clusters the proteins.

Part 1: Translations

The goal of this part is to minimize the distance between two structures by translating one of them with respect to the other. We accomplish this by equalizing their geometric centers. As discussed in class, computing optimal rotations (Part 2) requires a little more: the geometric centers must both be zero. This is done by the function `translate`:
function t = translate(s)
% t = translate(s). Translate a structure so that its geometric center
% is at the origin.
% s: [n 3]: n C-alpha coordinates
% t: [n 3]: translated structure
if size(s,2) ~= 3
    error('require 3-dimensional coordinates');
end
    t = s - repmat(mean(s,1), size(s,1), 1);

Part 2: Rotations

Assuming two length-n structures $A$ and $B$ are both translated to the origin, we can compute the rotation that minimizes their distance $d(A,B)$ (also called $D^2$ in class). As explained at length in class, this computation uses the method of Lagrange multipliers to relate the desired orthogonal matrix $U$ to the matrix $R$ defined by

$$ R = \sum_{i=1}^{N} r_i^B r_i^A $$

where $r_i^A, r_i^B$ are the coordinates of the $i$-th C-alpha atom of structures $A$ and $B$, respectively. If the SVD of $R$ is given by

$$ R = W S V^T $$

then the method yields eight candidates for $U$, given by

$$ U = \tilde{W} V' $$

where $\tilde{W}$ denotes the eight matrices that can be obtained from $W$ by negating some number of its columns. All eight possibilities for $U$ are orthogonal, but only four of them are proper rotations (i.e., only four satisfy $\det(U)=1$) and only one of these four minimizes $d(A,B)$. As explained in class, if a proper rotation $U$ can be obtained using a $\tilde{W}$ that involves no negations of any columns of $W$, then

$$ d(a,b) = \sum_{i=1}^{n} [(r_i^A)^2 + (r_i^B)^2] - 2 \sum_{k=1}^{3} (\sigma_1 + \sigma_2 + \sigma_3) $$

where the $\sigma_i$ are the elements of the diagonal matrix $S$. On the other hand, if this choice of $\tilde{W}$ results in $\det(U) = -1$, we know we can obtain the optimal proper rotation $U$ by choosing $\tilde{W}$ to be the matrix $W$ with the third column negated (because MATLAB orders singular values in decreasing order), and in this case we have

$$ d(a,b) = \sum_{i=1}^{n} [(r_i^A)^2 + (r_i^B)^2] - 2 \sum_{k=1}^{3} (\sigma_1 + \sigma_2 - \sigma_3) $$

The function $\text{optrot}$ computes the optimal rotation and the minimal distance:
function \([d, u, ss1] = \text{optrot}(s1, s2)\)

% \([d u ss1] = \text{optrot}(s1, s2)\). Compute the distance
% between two equal-length chains whose geometric centers are zero.
% 
% \(s1, s2: [n 3]\): input structures
% \(d: [1]\): distance between \(s1\) and \(s2\) after optimal rotation of \(s1\).
% \(u: [3 3]\): rotation matrix
% \(ss1: [n 3]\): result of rotating \(s1\) by \(u\)

\([w \text{ sing } v] = \text{svd}(s2' * s1); \% \text{SVD of } R\)

\(u = w * v'\); \% rot matrix

if \(\text{det}(u) < 0\) \% if \(u\) is not proper
    \text{sing}(end) = -\text{sing}(end); \% 'help svd' on 'decreasing order'
    \(w(:,\text{end}) = -w(:,\text{end});\)
    \(u = w * v'\);
end

\(d = \text{sum}(<\text{sum}(s1.^2 + s2.^2)) - 2 * \text{trace}(<\text{sing});\)

\(ss1 = (u * s1')';\)

% double-check. (Can't compute a relative error, since some
% distances may in fact be zero.)
if abs(d - \text{sum}(<\text{sum}((ss1 - s2).^2)))) > 1e-9
    \text{error('internal error')}
end

\(d = \sqrt{d};\)

---

**Part 3: The Distance Matrix**

Before we build the distance matrix, we need a piece of code that computes the distance between two proteins of different length. As explained in the project handout, this distance is defined as the minimal distance over all possible equal-length alignments, and it is computed by the function `protdist`: 
function [d, pp1, pp2, a, u] = protdist(p1, p2)

% [d pp1 pp2 a u] = protdist(p1, p2). Compute the distance between
% two proteins. If the proteins have different lengths, find the
% smallest distance among all possible alignments of the
% shorter one against the longer one.
%
% p1: [n1 3]: backbone of protein 1
% p2: [n2 3]: backbone of protein 2
% d, u: outputs of optrot for the optimal alignment
% pp1, pp2: p1 and p2 (or segments thereof) after translation and
%           rotation.
% a: [1]: index of the optimal alignment (see code)

n1 = size(p1, 1);
N2 = size(p2, 1);

if n1 >= n2
    plarge = p1;
    psmall = p2;
else
    plarge = p2;
    psmall = p1;
end

NL = size(plarge, 1);
NS = size(psmall, 1);

pps = translate(psmall);
d = Inf;

% Find min dist over all possible alignments
for i = 1:(nl - ns + 1)
    [dd uu ss] = optrot(translate(plarge(i:(ns+i-1),:)), pps);
    if dd < d
        d = dd;
        u = uu;
        ppl = ss;
        a = i;
    end
end

if n1 >= n2
    pp1 = ppl;
    pp2 = pps;
else
    pp1 = pps;
    pp2 = ppl;
end

We can now build our table (matrix) of distances, called distm. Here's the code:

nprot = length(proteins);
distm = zeros(nprot, nprot);
for i = 1:nprot
    for j = (i+1):nprot
        distm(i,j) = protdist(proteins{i}, proteins{j});
    end
end
distm = distm + distm'; % symmetrize for convenience
clear i j
And here’s the result:

\[
\begin{array}{cccccccc}
0 & 62.3124 & 116.4353 & 115.6811 & 62.5363 & 4.1161 & 198.1902 \\
62.3124 & 0 & 108.5739 & 102.2318 & 78.7225 & 62.3397 & 180.8759 \\
116.4353 & 108.5739 & 0 & 157.5985 & 113.0519 & 115.6084 & 189.4242 \\
115.6811 & 102.2318 & 157.5985 & 0 & 124.3862 & 115.5966 & 193.5639 \\
62.5363 & 78.7225 & 113.0519 & 124.3862 & 0 & 62.5803 & 192.4806 \\
4.1161 & 62.3397 & 115.6084 & 115.5966 & 62.5803 & 0 & 197.2393 \\
198.1902 & 180.8759 & 189.4242 & 193.5639 & 192.4806 & 197.2393 & 0 \\
\end{array}
\]

We can already see, in the seventh row (or column), that the 102L is considerably different than the other six proteins.

The function `plot_befaft` compares pairs of proteins before and after optimal rotation:

```matlab
function plot_befaft(p1, p2, name1, name2)

% plot_befaft(p1, p2). Plot two proteins "before and after" optimal rotations. If the proteins have different lengths, show only the common segment in the "after" picture.
% p1: [n1 3]: protein 1
% p2: [n2 3]: protein 2
% name1: string: name of p1
% name2: string: name of p2

[d pp1 pp2] = protdist(p1, p2);
p1 = translate(p1);
p2 = translate(p2);

figure
plot3(p1(:,1), p1(:,2), p1(:,3), 'o -', ... p2(:,1), p2(:,2), p2(:,3), '. -'); rotate3d
title(sprintf('%s and %s before optimal rotation', name1, name2)); legend(name1, name2);

figure
plot3(pp1(:,1), pp1(:,2), pp1(:,3), 'o-', ... pp2(:,1), pp2(:,2), pp2(:,3), '.-'); rotate3d
title(sprintf('%s and %s after optimal rotation. Distance = %f', ... name1, name2, d)); legend(name1, name2);
```

Now we plot and show three pairs, with “before” on the left and “after” on the right:

```matlab
plot_befaft(proteins{5}, proteins{6}, '2GDM', '1MBC')
plot_befaft(proteins{4}, proteins{5}, '4VHB', '2GDM')
plot_befaft(proteins{1}, proteins{6}, '1MBD', '1MBC')
```
Part 4: Hierarchical Clustering

We can now use the distance matrix to cluster the seven proteins into a hierarchy of clusters. We begin by making a conceptual distinction between a cluster and a clustering. Let \( P \) denote our original set of seven proteins. A \emph{cluster} is a subset of \( P \). A \emph{clustering} is a collection of clusters that form a partition of \( P \)—that is, the disjoint union of the clusters in a clustering must equal \( P \). For example,

\[
\{ \{1\text{MBD, 1SDK}_A, 4\text{VHB}_B\}, \{6\text{HBI}_A, 1\text{MBC, 102L}\}, \{2\text{GDM}\} \}
\]

is a clustering consisting of three clusters. In MATLAB, we will use an integer to represent a cluster, and we will represent a clustering using a length-7 row vector of
integers in which the \( i \)-th element is the cluster to which protein \( i \) belongs. The proteins are ordered according to the ordering given at the beginning. So the above clustering could be represented by the vector

\[
[1 \ 1 \ 2 \ 1 \ 3 \ 2 \ 2]
\]

An equivalent representation is

\[
[6 \ 6 \ 1 \ 6 \ 8 \ 1 \ 1]
\]

The actual integers used are immaterial. We require only that different clusters should have different integers.

We need a function that computes the distance between two clusters of a clustering, as defined in the project handout. The function \texttt{clustdist} computes this distance in one line:

```matlab
function d = clustdist(cc, i, j, dmat)
    % d = clustdist(cc, i, j, dmat). Compute the distance between two clusters, given a distance matrix. The cluster distance is defined to be the average of inter-cluster distances.
    % cc: [1 n]: clustering of n objects
    % i, j: [1]: two clusters in cc
    % dmat: [n n]: distance matrix
    % d: [1]: distance between clusters i and j, according to dmat.
    d = mean(mean(dmat(find(cc==i),find(cc==j))));
```

Next we have the function \texttt{minclustdist}, which finds the two closest clusters in a clustering:
function [cm1, cm2, dmin] = minclustdist(cc, dmat)

% [cm1, cm2, dmin] = minclustdist(cc, dmat). Find the nearest pair of
% clusters in a clustering.
% cc: [1 n]: clustering of n objects
% dmat: [n n]: distance matrix
% cm1, cm2: [1]: closest elements of cc
% dmin: [1]: distance between c1 & c2

clusts = unique(cc);
n = length(clusts); % number of clusters

dmin = Inf;
for i = 1:n
    for j = (i+1):n
        c1 = clusts(i);
        c2 = clusts(j);
        d = clustdist(cc, c1, c2, dmat);
        if d < dmin
            dmin = d;
            cm1 = c1;
            cm2 = c2;
        end
    end
end

We need a function to merge two clusters:

function cc = merge(cc, i, j)

% cc = merge(cc, i, j). Merge two clusters of a clustering.
% cc: [1 n]: clustering of n objects
% i, j: [1]: two clusters in cc
% cc (out): [1 n]: merged clustering

ccc = cc == i = j;

Finally we can implement the clustering algorithm, which reads almost exactly like its
description in the project handout:
function h = clust(dmat)

% h = clust(dmat). Cluster a distance matrix hierarchically
% dmat: [n n]: distance matrix between n objects (assumed
% symmetric).
% h: [n n]: clustering hierarchy. The i-th row gives the clustering
% at the start of the i-th iteration.

n = length(dmat);
h = 1:n;
while length(unique(h(end,:))) > 1
    [c1 c2] = minclustdist(h(end,:), dmat);
    h = [h; merge(h(end,:), c1, c2)];
end

And the final result is:

```
>> h = clust(dist)

h =
    1  2  3  4  5  6  7
   6  2  3  4  5  6  7
   6  6  3  4  5  6  7
   6  6  3  4  6  6  7
   6  6  6  4  6  6  7
   6  6  6  6  6  6  7
   7  7  7  7  7  7  7
```

In picture form:

As expected, 102L is evolutionarily remote from the other six proteins.