CSCI 2951G: Assignment 3

Paul Valiant

September 21, 2012

Due: Wednesday, October 10.

Turn in: A file `myloadprotein.m`, along with any data files you created or edited that are needed to run it. Also, turn in a writeup that describes the design of your code and any interesting features of it, or of the process that led you to it; if there are particular examples that show off how good your code is, include them in the writeup. Also be sure to credit others (from the internet or from class) for ideas you used.

1 The Problem:

Having an intuitive framework for how to arrange the atoms of a chain of amino acids in a realistic (that is, low-energy) way is important from many perspectives of the protein folding challenge. The code I distributed does not do a particularly good job of it, and your challenge in this homework is to do a (substantially) better job.

For example, if you load a chain of amino acids, as we have been doing, and then compute (via the `proteins` function) what the initial potential energy of the system is, you will likely find a number that is at least several thousand:

```
d=loaddat;d2=loadprotein(d,{'asn','leu','tyr','ile','gln','trp','leu','lys'},'b');
p=loadphys;p2=applyphys(d2,p);
p2.output=1;p2.pause=0;
out=proteins(d2.p,d2.edges,d2.types,p2);
sum(out.GGB(1,2:end))
```

This returns an energy of 3847.3. However, when left alone for even the tiniest amount of time, the system's energy becomes much smaller:

```
p2.output=100;p2.pause=0;
out=proteins(d2.p,d2.edges,d2.types,p2);
sum(out.GGB(end,2:end))
```

This returns an energy of -404.8, and, clearly, we could minimize even further with better parameters:

```
p2.output=1000;p2.pause=0;p2.params=[0 0 .02]
out=proteins(d2.p,d2.edges,d2.types,p2);
sum(out.GGB(end,2:end))
```

This returns an energy of -615.7. Remember that k_BT in our units is roughly 0.6, so each 0.6 that we decrease the energy by makes the configuration a factor of $e \approx 2.71828$ more likely, so these improvements are drastic.

The goal of this assignment is to create a function `myloadprotein` that takes as input a list of amino acid names, and outputs a 3-column matrix with each row specifying the position of one of the atoms in the molecule. The energy of your proposed positions for the atoms can then be evaluated by inputting this matrix as the first argument of the `proteins` function:

```
d=loaddat;d2=loadprotein(d,{'asn','leu','tyr','ile','gln','trp','leu','lys'},'b');
p=loadphys;p2=applyphys(d2,p);
p2.output=1;p2.pause=0;
betterpositions=myloadprotein({'asn','leu','tyr','ile','gln','trp','leu','lys'});
out=proteins(betterpositions,d2.edges,d2.types,p2);
sum(out.GGB(1,2:end))
```

Your goal is to write a function that yields as low an energy as possible, by leveraging intuition and your knowledge about proteins. Your function is *not* allowed to simulate physics as we did in the above examples, but instead should provide a shortcut that yields almost-as-good results, without the computational effort. This is in line with the overall goals of this course: investigate how we can develop new algorithms to fold proteins which bypass the enormous effort of simulating physics femtosecond-by-femtosecond.

You should aim to get the energy below 0, as a starting point.

1.1 Ground rules:

Your function must work for any sequence of amino acids, but it is allowed to fail or crash on sequences that include terminal amino acids (the ones with four letter abbreviations that start with "c" or "n"), and also proline ("pro") which has a different structure than the others.

Note that the order of atoms in your matrix must match the order that `loadprotein` produces, so that the calculation of energy interprets each row of the matrix correctly. Explicitly, each amino acid is stored in sequence, and within each amino acid, the atoms are stored in the order of the data file ("amino12.in"), which is also the order that the atoms are stored in the cell array `d`, for each amino acid.

You can use implicit water or not; it should not make much difference unless your code is *really* good.

Your function is *not* allowed to run `proteins` in any form. (Though, of course, when designing your function, you should do extensive experiments with `proteins`.)

You have a choice of whether to start with the results of `loadprotein` and refine them, or to construct your atom positions from scratch.

Your function should be relatively quick to run.

1.2 Useful Matlab snippets:

1.2.1 Snippet 1:

Each of the amino acids, as loaded into the variable `d` by the `loaddat` command is arranged reasonably well in space, so one tactic is to just treat each amino acid as a rigid object, and simply try to find ways to translate and rotate each amino acid into an appropriate place on the chain. For example, you could fill in the following code outline (copy this to a file `myloadprotein.m` and save it):

```
function out=myloadprotein(seq)
d=loaddat; out=[];
names=[]; for i=1:length(d), names{i}=upper(d{i}.name); end;
for i=1:length(seq)
    loc=strmatch(upper(seq{i}),names);
    SHIFT= ??? ;  ⟵ Fill this in with a 1 × 3 vector to shift the current amino acid by.
    ROTATE= ??? ;  ⟵ Fill this in with a 3 × 3 rotation matrix to rotate the current amino acid by.
    out=[out;d{loc}.p*ROTATE+repmat(SHIFT,size(d{loc}.p,1),1)];
end
```

This code assembles a list of the names of all the amino acids loaded by `loaddat` and stores it as `names`; then for each amino acid in the input sequence, it searches for it in the list of names and stores its index in `loc`. (The Matlab command `upper` changes everything to uppercase, so that we do not need to worry about cases.) Note that `d{loc}.p` is now a matrix storing the positions of all the atoms in the named amino acid. We can transform this by a 3×3 rotation matrix via just a matrix-matrix multiplication, which in Matlab can be done simply with the standard multiplication operator. Next we want to add the `SHIFT` to each of the atoms, which, if the amino acid has n atoms, requires copying the `SHIFT` vector n times vertically. Given a matrix M, Matlab can replicate it n times in the vertical direction (and only 1 time in the horizontal direction) with the command `repmat(M,n,1)`, which we do here, after figuring out the number of rows needed by measuring the size of the first dimension of `d{loc}.p`.

If you want to just get the code running, you could try setting `SHIFT = [3*i 0 0];` to shift the ith amino acid by $3i$ Angstroms in x direction, and `ROTATE = eye(3);` to rotate by the identity matrix (not rotating at all) – `eye(n)` generates the $n \times n$ identity matrix; for $n = 3$ this is equivalent to `[1 0 0;0 1 0;0 0 1]`.

For those of you not familiar with linear algebra, this approach might take some extra effort. Some useful Matlab commands include: the single quote `'` is the transpose operator in Matlab, which flips rows and columns of a matrix (technically it is the conjugate transpose, but as we are not dealing with complex numbers, this is the same thing); thus a matrix `M` is orthonormal if `M*M'` equals the 3×3 identity matrix, `eye(3)`; given two 3-dimensional vectors, their *cross product* is a vector that is orthogonal to both of them, found in Matlab with the `cross(a,b)` command; more generally, given 3 vectors in 3 dimensions as the columns of a 3×3 matrix `M`, the matlab command `[q,r]=qr(M);` computes a matrix `q` whose first column has length 1 and is in the direction of the first column of `M`, whose second column has length 1 and is orthogonal to its first column but is in the span of the first two columns of `M`, and whose third column has length 1 and is orthogonal to the other two. (Note that this Matlab command has two outputs, `q` and `r`, and it will mean something *different* if you try to run it with only one output, as `q=qr(M)`.) In Matlab, `sin` and `cos` expect arguments in radians, but `sind` and `cosd` expect arguments in degrees.

1.3 Snippet 2:

An alternative approach to manipulating the positions of atoms is to start with the positions as loaded by the provided `loadprotein` function and then modify them. One biochemically natural way to do this is, instead of modifying the positions of the atoms, to instead consider each atom's position as being specified by the triple (distance, angle, dihedral angle), where the distance and angles are specified with respect to three other atoms. (As long as there are no circular definitions and no three atoms are collinear, this is well-defined).

There are two Matlab functions in the `proteins` package that convert from position to this structural notation, and then back again. Given atom positions in a 3-column matrix `pos`, you can compute the structure with `str=findstruc(pos,d2.struc,0);` – the third argument is never used, but tells the `findstruc` function to generate 3 dummy atoms at the beginning of the structure, which are needed to unambiguously recover the position from the structure. In the other direction, you can reconstruct positions from a structure as `pos2=myang2pos(str);` which should yield something very close to the original pos, if you did not modify the structure in the interim. The structure is represented as a 6-column matrix, where the first three columns name the indices of the three atoms that the current atom's position will be defined in terms of; the fourth column describes the distance from the atom in the first column; the fifth column describes the angle formed by the current atom, the atom in the first column, and the atom in the second column; the last column describes the dihedral angle formed by the current atom and the atoms in the first three columns in order. Note that you might find it easier to play with the structure if you change which atoms each atom is defined by. This suggests the following structure for your code:

```
function out=myloadprotein(seq)
d=loaddat; d2=loadprotein(d,seq,'b'); ⟵ Possibly modify d2.struc now
str=findstruc(d2.p,d2.struc,0); ⟵ Now modify the last 3 (2?) columns of str intelligently
out=myang2pos(str);
```

1.3.1 Analysis tools:

In Lecture 4 we saw a code snippet that figures out which atoms have the highest forces on them, and displays them "big" in `proteins`.

Run one timestep of `proteins` with atom positions specified by `pos`:
`p2.output=1;p2.pause=0;out=proteins(pos,d2.edges,d2.types,p2);`

Then, for some threshold, say 1000, set `big` to be those atoms whose forces are bigger than 1000:

```
p2.big=find(sqrt(sum(out.f.^2,2))>1000);
```

And run `proteins` again with `p2.output=0; p2.pause=1;` to see what you have.

Also, remember that the potential energy output by `proteins` is actually specified as 6 numbers, the 2nd through 7th columns of the output's `GGB` field. (If the 8th column is also nonzero, please comment out the last line in `applyphys.m`; the `dihedralbump` function is an experimental feature that should be disabled.) These 6 types of potential energy report, respectively, 1) the energy from the implicit water model (which will be 0 if you are not using implicit water), 2) energy from the electrostatic term of the force field, 3) energy from the Lennard-Jones term, 4) energy from the bonds being stretched from their equilibrium length 5) energy from the angles being stretched from their equilibrium angle, and 6) energy from the dihedral angle term. In particular, when you start out, the Lennard-Jones term will probably be far bigger than the other terms, indicating that certain atoms are "overlapping" each other, which has an energy penalty proportional to $\frac{1}{r^{12}}$, which can be huge.

1.3.2 Data files in Matlab:

If you end up tweaking intricate data by hand, you might want to save the results to disk so that your code can load the data next time it runs. The Matlab `save` and `load` commands are useful for this: to save variables `var1` and `var2` in the file `myfile.mat`, type
`save myfile var1 var2`

You can load these variables with `load myfile`

1.3.3 Debugging in Matlab:

Matlab has an interactive debugger, with the general features you would expect. Perhaps the most useful is `dbstop if error`, which, after you type it means that next time your code crashes, it pauses in the debugger and lets you inspect all the local variables, etc. You can revert this setting with `dbclear all`. You can see where in the code you are with `dbstack`, and can examine variables from different levels of the stack with the `dbup` and `dbdown` commands. Standard debugging tools for stepping through code and enabling breakpoints are available in the Matlab editor. You can see the memory usage of local variables by typing `whos`, and if your code is running too slowly, you can profile it with the visual profiling tool (which is quite powerful), by typing `profview`.