CSCI 2951-G: Computational Protein Folding Fall 2012

Lecture 2 — 09/10/2012

Prof. Paul Valiant *Scribe: William Poole*

Simulating Physics

The Energy Function

The physics of protein folding can be formulated relatively simply with the concept of energy. The total energy of the system E can be written as a function of the position vector $\vec{p} = (p_1, p_2, ...p_N)$ and the velocity vector $\vec{v} = (v_1, v_2, ...v_N)$, where the subscript of the components denotes the particle index (and in total there are N particles). More specifically, the energy of the system can be divided into kinetic energy $T(\vec{p}) = \sum_i \frac{1}{2} m_i v_i^2$ and potential energy, $U(\vec{p})$, which in the case of protein folding can be approximated by the complicated function seen in the previous lecture. To reiterate,

$$E(\vec{p}, \vec{v}) = U(\vec{p}) + T(\vec{v}). \tag{1}$$

The force, $\vec{F}$, can then easily be determined using the equation

$$\vec{F}(p) = -\nabla U(p), \tag{2}$$

where ∇ denotes the gradient.

An Iterative Approach

The dynamics of protein folding can be simulated in a relatively simple iterative manner where the position and velocity vectors are updated based upon the following rules,

$$\vec{p} \leftarrow \vec{p} + \vec{v}\Delta t \tag{3}$$

$$\vec{v} \leftarrow \vec{v} + \vec{F}(p)\Delta t \tag{4}$$

where "$\leftarrow$" denotes that the variable on the left hand side gets assigned the value of the expression on the right hand side, and Δt denotes the timestep, namely the amount of time we aim to simulate in each iteration. This approximation gets steadily more accurate as Δt approaches 0. In general, calculating the force as a function of the position vector is the most computationally intensive part of a simulation.

Example: A One Dimensional Spring

To illustrate the above simulation scheme, consider the very simple case of a linear one dimensional spring. In this case (taking all constants to one), $F(p) = -p$ which is equivalent to the potential energy function $U(\vec{p}) = \frac{1}{2}p^2$. The equations (3) and (4) can then be written,

$$p \leftarrow p + v\Delta t \tag{5}$$

$$v \leftarrow v - p\Delta t \tag{6}$$

The way these rules are evaluated is very important to the success of the simulation. As a first example, consider evaluating (5) and (6) simultaneously, namely evaluating both right hand sides at the same time, and then updating p and v simultaneously. Take $\Delta t = 1$ with the initial condition $(p, v) = (1, 0)$. Figure 1 illustrates the time evolution of the system simulated this way. The Matlab code used to generate this graph is

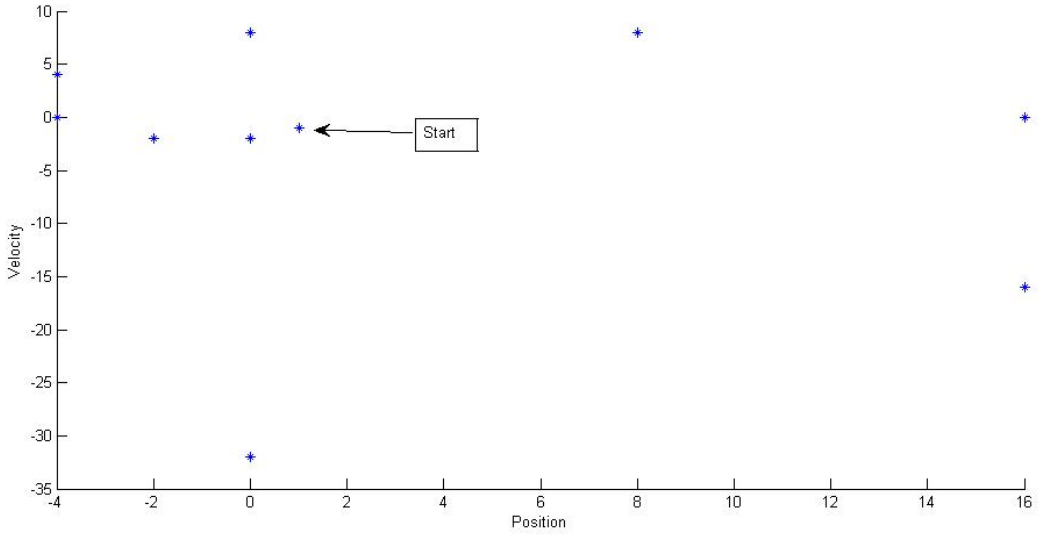

Figure 1: Simultaneous Evaluation of p and v

p=1;v=0; dt=1; figure(1);clf;hold on; for i=1:10; p1=p+v*dt; v1=v-p*dt; p = p1; v = v1; plot(p,v, '*'); end

Matlab explanation: `figure(1)` creates a new figure, labeled "1"; `clf` clears and resets the figure; `hold on` makes it so that everything plotted is retained, instead of only the most recent plot command showing up; `1:10` creates the vector `[1 2 3 4 5 6 7 8 9 10]` and the for loop iterates through it; the `'*'` at the end of the plot command plots stars at each point.

As can be seen in figure 1, the simultaneous evaluation of (5) and (6) results in a spiral evolution of the system where the energy increases indefinitely (the spring's extension increases and the magnitude of velocity increases). Explicitly, each iteration here rotates the point 45 degrees around the origin and *scales* it by a factor of $\sqrt{2}$, leading to exponential blowup. Clearly, this form of numerical integration is not physically realistic. However, a simple modification - namely evaluating (5) then (6), by evaluating (6) based on the *new* value of p just computed by equation (5) results in a more realistic simulation of the physics. (Alternatively, the other order works equally well: evaluating equation (6) and then (5).) This scheme is called the *leapfrog* method. The following graph illustrates this schema, again taking $\Delta t = 1$ with the initial condition $(p_0, v_0) = (1, 0)$. The code used to generate this graph is similar to the previous code except that the data points were stored in an array called data and plotted together, so a line could be drawn along with the graph.

p=1;v=0; dt=1; dat=zeros(100,2); figure(1);clf; for i=1:100; p=p+v*dt; v=v-p*dt; dat(i,1)=p;dat(i,2)=v; end; plot(dat(:,1),dat(:,2),'*-'); axis equal

Matlab explanation: `zeros(100,2)` creates a matrix that has 100 rows and 2 columns; `dat(i,1)` accesses the `i`th row and 1st column of `dat`; `dat(:,1)` returns the entire first column of `dat`; the `'*-'` at the end of the plot command plots both stars at the points, and lines between the points; `axis equal` makes the x and y axes haveequal scale (so that circles look circular instead of stretched).

As can be seen in figure 2, something akin to energy is conserved since the motion of this system is periodic. If energy were conserved, then the points would all lie on a circle around the origin, but as is, they lie on something "almost" a circle. The reason for this is that this integration scheme is *symplectic*, an important term which will be covered next lecture.

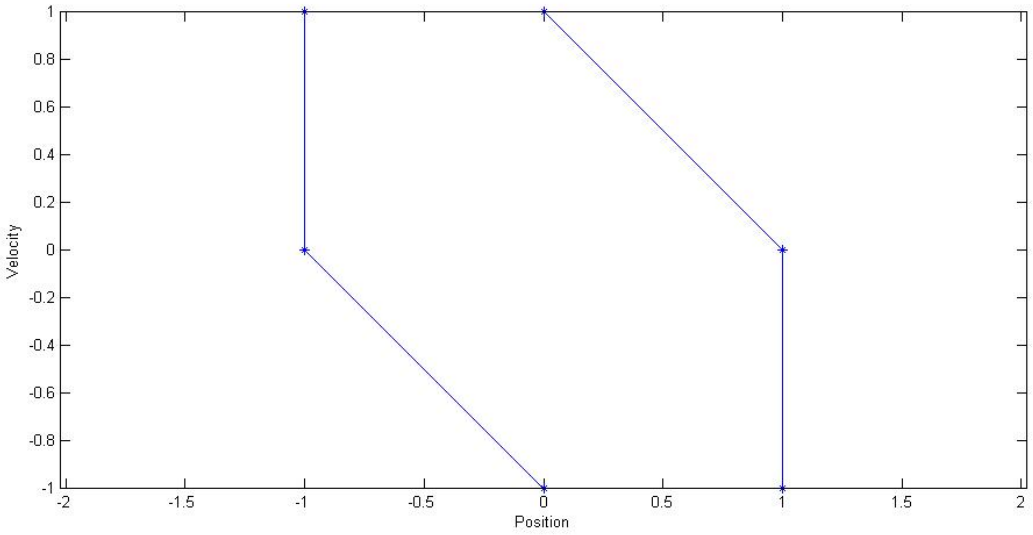

Figure 2: Non-Simultaneous Evaluation of p and v

Finally, consider what happens as Δt is changed. The smaller Δt becomes, the closer the simulation comes to the actual physics being modeled (in this case a circular trajectory on the position-velocity graphs). However, as Δt increases, the skewness of this trajectory also increases, ultimately blowing up at the value $\Delta t = 2$. This effect is illustrated in figure 3.

Back to protein folding

The considerations above about a maximum timestep for a simple spring system are the underlying reason why protein folding must be simulated in femtoseconds. More specifically, bonded hydrogen atoms oscillate with a period of approximately 10^{-14} seconds. Since the above analysis of a spring with period 2π showed that our method blows up for timesteps greater than 2, but starts looking reasonable for timesteps around 1, we can divide through by $2\pi \cdot 10^{14}$ to see that for hydrogen atoms, it might be reasonable to use timesteps of 1.5-2.0 fs. Even so, this value of Δt is not entirely realistic from a physical point of view, as the difference between the red and green paths in figure 3 shows. However, as we will discuss in the next lecture, the fact that we are using a symplectic simulation scheme means that, instead of conserving energy E, we are conserving a nearby but different function, $E_{\Delta t}$, which converges to E as Δt converges to 0. As we discussed in the first lecture, the energy function E that we use for protein folding is only a rough approximation of the true energy of a protein, E_{true}. Thus what our numerical methods will actually conserve, $E_{\Delta t}$ is roughly equal to E, the expression from Lecture 1, which is roughly equal to E_{true}, the true value. Since we're already approximating energy, the second approximation from numerical integration with a large timestep is just "more of the same", and thus large timesteps generally do not hurt protein folding simulations – provided the timestep is still small enough to stay away from blowup behavior.

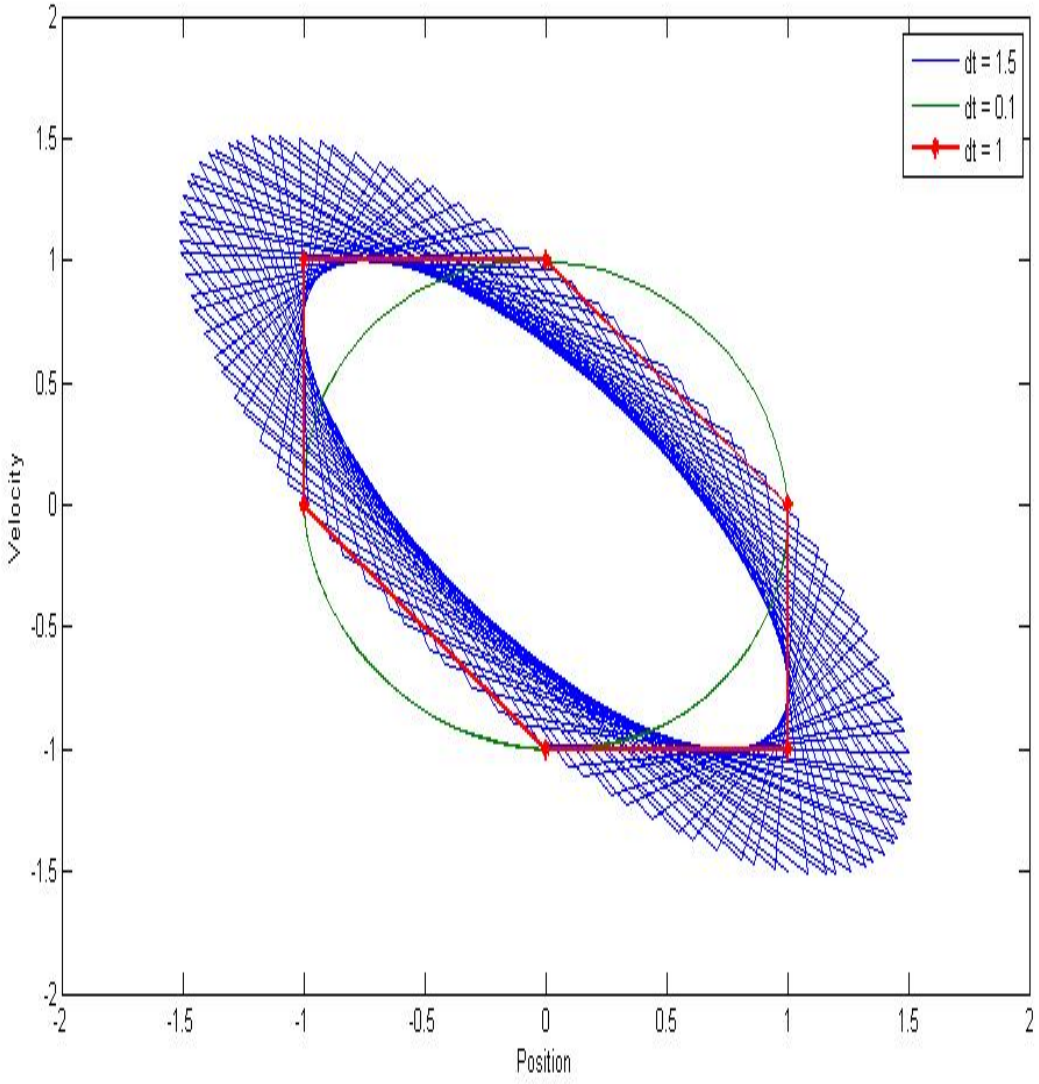

Figure 3: As Δt decreases, the trajectory becomes circular