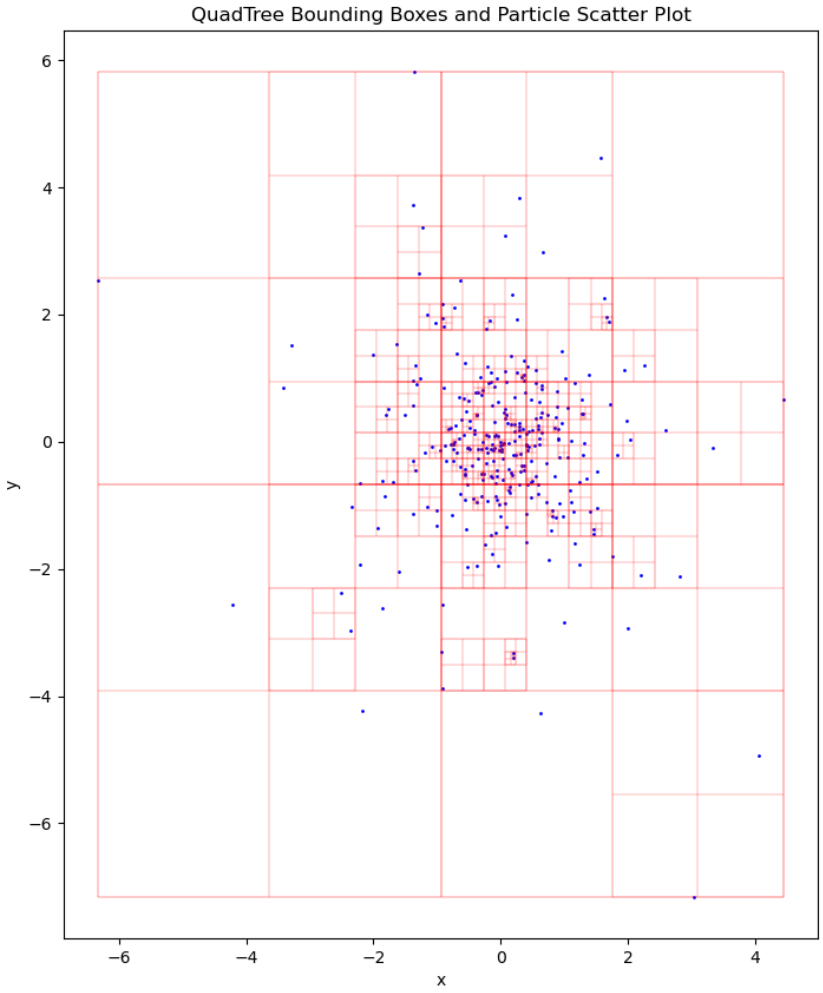
N-body simulations using the Barnes-Hut algorithm

David Bruijne
14608278

May 28, 2024



1 Introduction

In computational physics, the runtime of simulations plays an important role financially. A fast but less accurate model could be more enticing than a slow and accurate model. N-body problems can be especially difficult since the complexity scales as $\mathbb{O}[n^2]$.
The Barnes-Hut algorithm allows computational physicist to get the best of both worlds, speeding up the runtime while minimizing the loss of accuracy. It does this by approximating the force of gravity between distant clumps of particles using their center of mass and total mass instead of calculating the forces for the individual particles. Using this technique, the complexity can be reduced to $\mathbb{O}[n\log(n)]$. The approximation is done by a 'resolution parameter' θ. The higher this parameter, the more often it will use the center-of-mass approximation. The lowest (useful) value, $\theta = 0$ corresponds to the true, direct summation, of the forces acting on the particle. We introduce the 'smoothing parameter' ϵ because when the distance between two particles shrinks the force (and thus the acceleration) will blow up to infinity. The smoothing parameter basically sets an upper limit to the maximum force a particle can experience by another one.
In this review I will document my attempt to create such an algorithm. First I will describe how the code works and how to run such simulations using my code. Then, I will show you how it compares to the direct summation method And lastly I will talk about possible improvements that can be made. In the appendix I've added some interesting simulations I've made using the code.

2 The code

2.1 Choice of Integration

The choice of integrator to use has implications for both runtime and overall error. I've chosen to mainly use the leap-frog integrator as it naturally conserves energy [1]. The error size on the leap-frog integrator is of order $(dt)^3$ for the position and of order $(dt)^2$ for the velocity. For comparison I've also implemented the Runge-Kutta-4 (RK4) integrator, using both adaptive and non-adaptive timesteps. These integrators are more precise but are not naturally energy conserving. Also we need to recalculate the force on each particle more often, so I expect it will be a slower method of integration. In this rapport I will perform a quick error analysis and a runtime analysis to compare these different integrators to each other.

2.2 How to run the code

The code is build upon the skeleton code provided and can be found in appendix A. It contains three classes and multiple functions. The first class, the particle, represent the properties of the particles. Here the position, the velocity and the mass of the individual particles are stored. The QuadTreeNode class represent the individual nodes contained in the full tree. Lastly, the Quadtree class represents the entire tree.
To run a simulation we first need to create some particles. We can do this in multiple different ways.

```
particles = initialize_particles(N, 0.01,
                                 Speed_type= 'Differential',
                                   G = G,
                                     pos_mass = [0, 100])
planets_and_sun = Solar_system(G)
```

The 'initialize_particles' function generates N random particles, the second argument determines the mass; in this case 0.1. The way we generate the initial velocity can be determined with the optional argument 'speed_type'. If we don't specify this it will randomly generate a speed. There are two speed types we can give as input, 'Differential' and 'Rotational'. Note that we must also give a 'G' term if we use the 'Differential' option. The last optional argument is 'pos_mass= $[r, M]$'. This is option adds a particle at radial coordinate r with mass M to our list of particles before we generate the N random particles, allowing us to use the 'Differential' speed generation to simulate particles rotating around a heavy object.

Another way we can initialize the particles is using the 'Solar_system' function. This only takes in the Gravitational constant G and returns a list of particles. representing the solar system. For this function to run we set G to $4\pi^2$, the space, mass and time units will then be in AU,$M_\odot$ and years.

Now that we have some particles we can start building the tree. We use the Quadtree object which takes in a list of particles, the resolution parameter θ and a maximum recursion depth. Then we call the .insert() method to create the tree.

```
quadtree = QuadTree(particles, theta, max_depth)
quadtree.insert()
```

If we want to insert a particle after creating the tree, we can call the 'insert_particle' function:

```
insert_particle([-x,y], [vx,vy], mass)
```

This function takes 3 arguments, the position and speed (as a list with single (float) values) and a mass (float). The difference between this function and the optional argument 'pos_mass= $[r, M]$' in the 'initialize_particles' function is: This function inserts the new particle after creating the tree (and thus doesn't affect the creation of the particles) while the 'pos_mass= $[r, M]$' adds this particle first and then creates the rest. One can be used to create particles rotating around a central mass (the velocities of the particles are depended on the mass they're rotating around). The other one for example, can be used to create a comet-like object passing through the solar system.

Simulating particles is done by two functions.

```
animate_particles = get_animate_particles(quadtree,'RK4', G, eps, dt)
ani = FuncAnimation(plt.gcf(), animate_particles,
                    interval=1, frames=300, repeat=False)
plt.show()
```

The 'get_animate_particles' function takes in 6 arguments;

- Quadtree object; The tree we want to simulate
- Integration method; either 'RK4' or 'RK4 adaptive', If not specified it will use leap-frog integration.
- (optional) G = 1; the gravitational constant to be used
- (optional) eps = 0.001; the smoothing parameter of the Barnes-Hut algorithm
- (optional) dt = 0.01 ; the time-step to take
- (optional) with_boxes = False; A boolean that triggers the visibility of the boundary boxes of the nodes

The duration of the simulation can be increased or decreased by tweaking the 'frames' argument in the 'FuncAnimation' function

We can make a scatter-plot with all the particles and boundary boxes using the 'scatter particles and bboxes' function with as input our quadtree we want to visualize.

```
scatter_particles_and_bboxes(quadtree)
```

We end up with a figure like the on the front page of this rapport. The formatting of the plot is hard-coded but can be changed if tinker with the function. The same also holds for the animation.

Other functions that are defined are used throughout the code but understanding these functions is not necessary to create simulations. For the creation of some figures, I copied the code into a notebook and made a quick for loop instead of creating separate functions. This part of the code is found in the second appendix B.

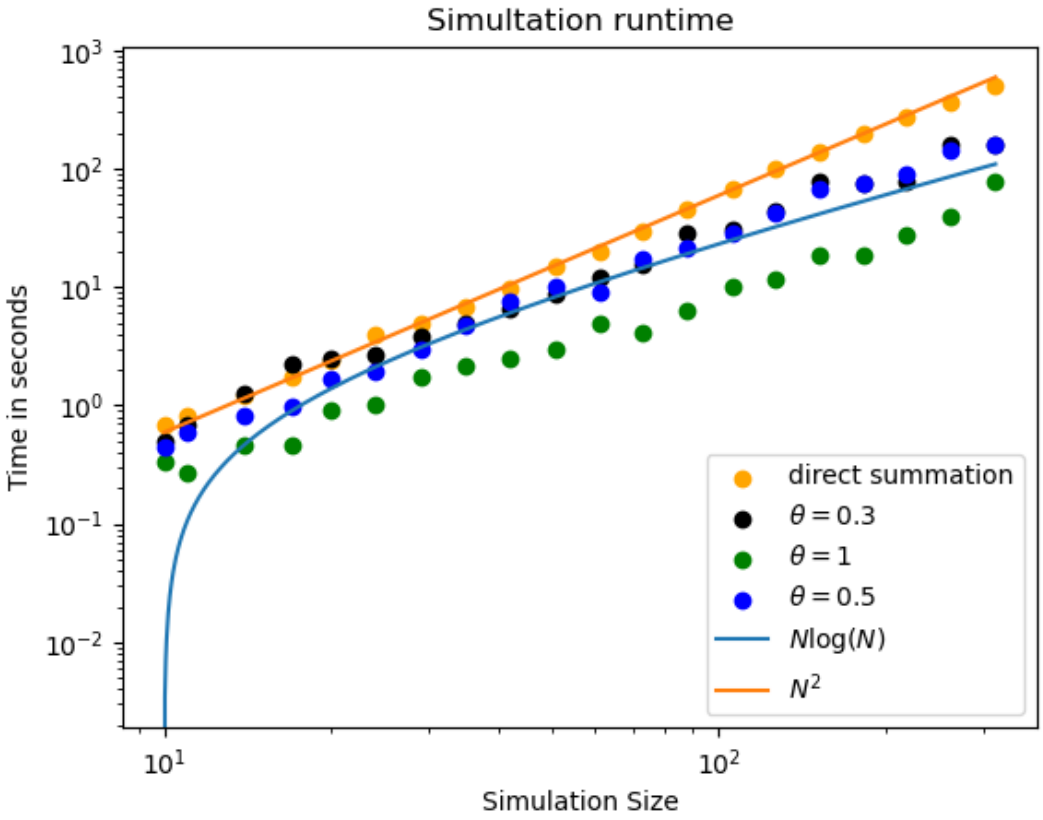


Figure 1: Runtime versus simulation size. On the x-axis we see the size of the simulation in Log-scale. On the y-axis we see the runtime in seconds, also in Log-scale. These data-points are measured using 100 time-steps with the Frog-leap integrator. The figure shows 4 different values for θ with whom the run-times were measured (0 corresponding with 'direct summation'). The blue and orange line represent how the direct summation and the Barnes-Hut algorithm should scale as N increases. Note that this has been fitted by eye only.

3 Results

3.1 Resolution parameter

The resolution parameter θ controls whether for a given particle we use direct summation or approximation. The approximation is done by the center of mass and the total mass of the node in question. Setting $\theta = 0$ we use direct summation all the time, The higher θ the more we approximate.
In figure 1 we see the run-speed plotted for different simulation sizes given different values for θ. As you can see the 'direct summation method' generally takes the longest amount of time, also by eye, the slope is around 2. This is what we expected as the computational complexity scale with N^2. Generally the fastest simulation was $\theta = 1$, however using these data-points we can't say for sure if it scales with $N \log(N)$ even tough it could very well be the case. Since the runtime using ≈ 300 particles took around 100 seconds to run I've decided to not expand this figure any further.

θ, the resolution parameter also affects the error on the energy. The energy, which is calculated through direct summation, seems to be more stable for lower values of θ as we can see in figure 2. This is to be expected since for lower θ the algorithm will use center-of-mass approximation less and less. The reason why so many values for θ still overlap could be explained by the size of my simulation and by my radial distribution. But I can't say for certain.

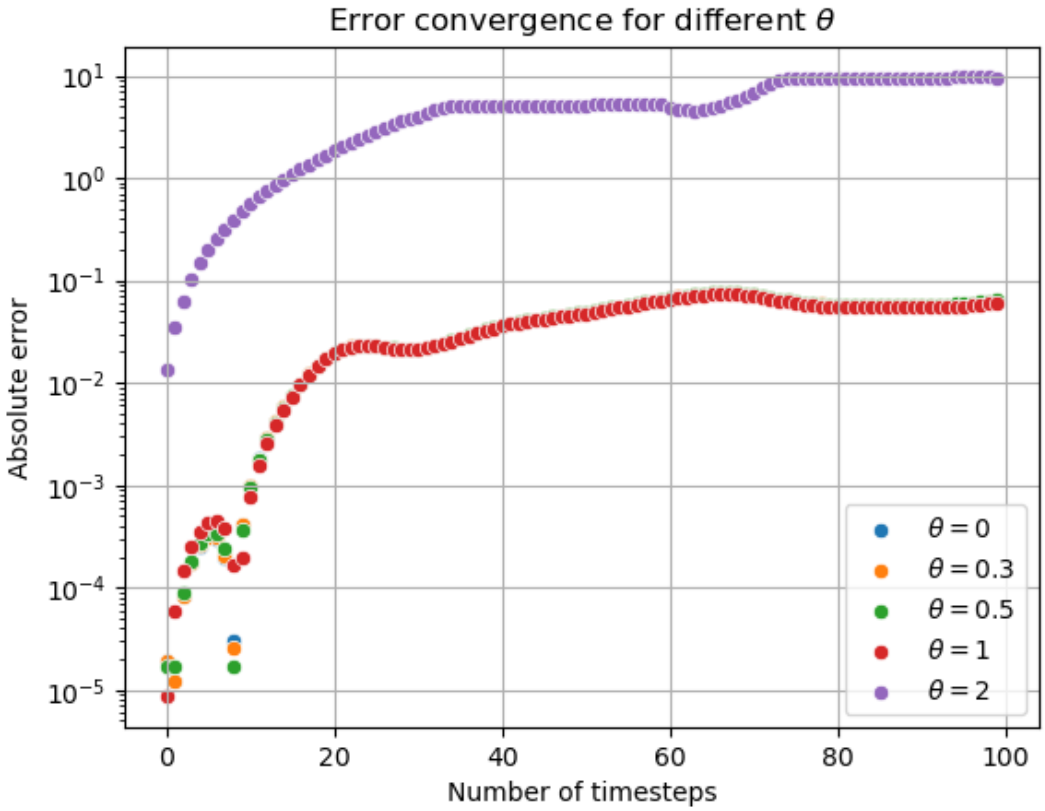


Figure 2: In this figure we see the behaviour of the energy error over time fir different θ. I simulated the same six particles for 100 time steps using the frog-leap integrator with $\epsilon = 0.1$ and $dt = 0.001$. Most values for θ overlap, except for $\theta = 2$

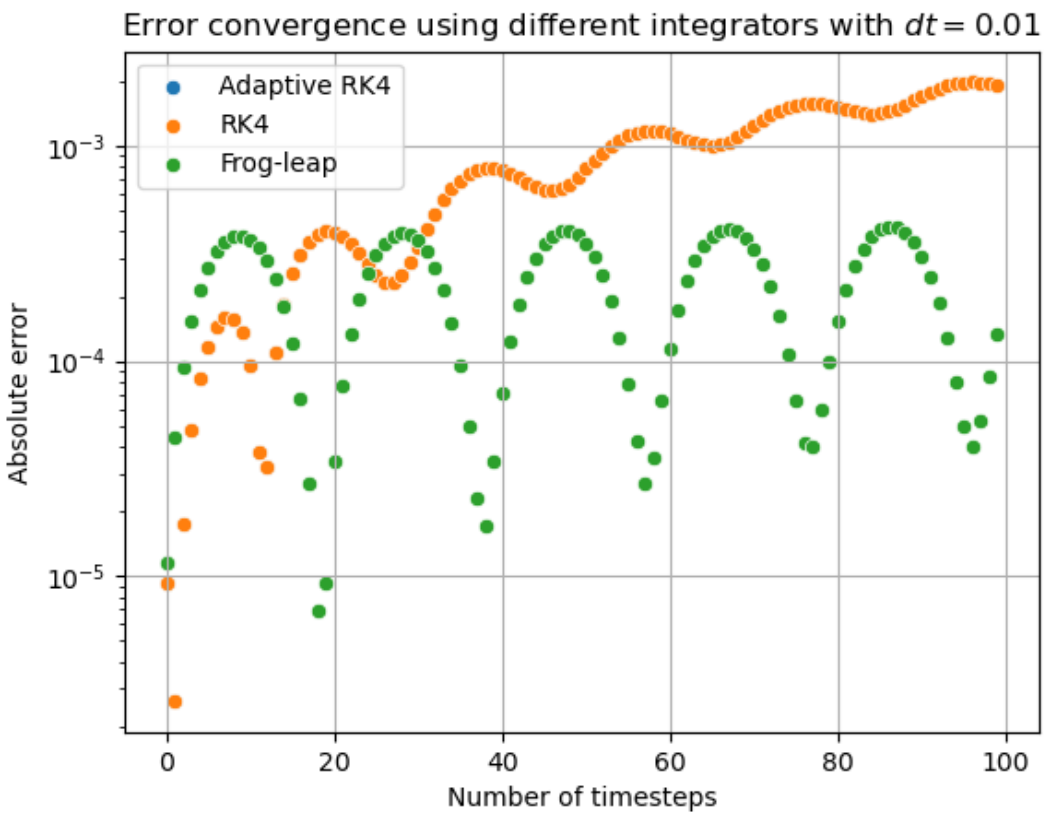


Figure 3: In this figure we see the behaviour of the energy error over time. I simulated the same six particles for 100 time steps, with $\theta = 0$, $\epsilon = 0.001$ and $dt = 0.01$. In blue we see the adaptive RK4 which is overshadowed by the regular RK4. In orange we see the Frog-leap integrator.

3.2 Integration methods

Energy should be conserved in our simulation. The choice of hyper-parameters, integrator to use and the starting positions of the particles heavily influences the behaviour of the total energy of the system. In 3 we see how the energy evolves over time. In this example, The frog-leap integrator seems to perform the best. The adaptive RK4 should perform better than the regular RK4, since it doesn't it leads me to suspect that I've implemented it wrong. It does change the the size of the time-step so it could also be due to the choice of hyper-parameters. While playing around with my code I've found that the leap-frog integrator generally outperforms the other two implemented integrators (as in, it is more stable), as expected. The RK4 should outperform the leap-frog integration per time-step but since the RK4 error slowly drifts upward in our example while the leapfrog doesn't this is hard to see. Looking at the first few data points we see traces of this. In between timesteps the leap-frog makes bigger jumps than the RK4 method, suggesting that the error between timesteps is bigger.

In figure 4 we see the (time)performance for each integrator, and as expected, the frog-leap integrator is the fastest followed by the non-adaptive RK4 integrator. By eye we see that the adaptive RK4 is thrice as slow as the regular RK4 integrator. This is because we perform three integrations using the RK4 method in that integration scheme. The difference between the RK4 and the leap-frog method can be explained in a similar way. The bulk of the calculations are done when computing the forces acting on each particle. In the RK4 method we do this four times, compared to twice in the leap-frog method. We thus expect the leap-frog method to be roughly twice as fast and this is (by eye) what we see.

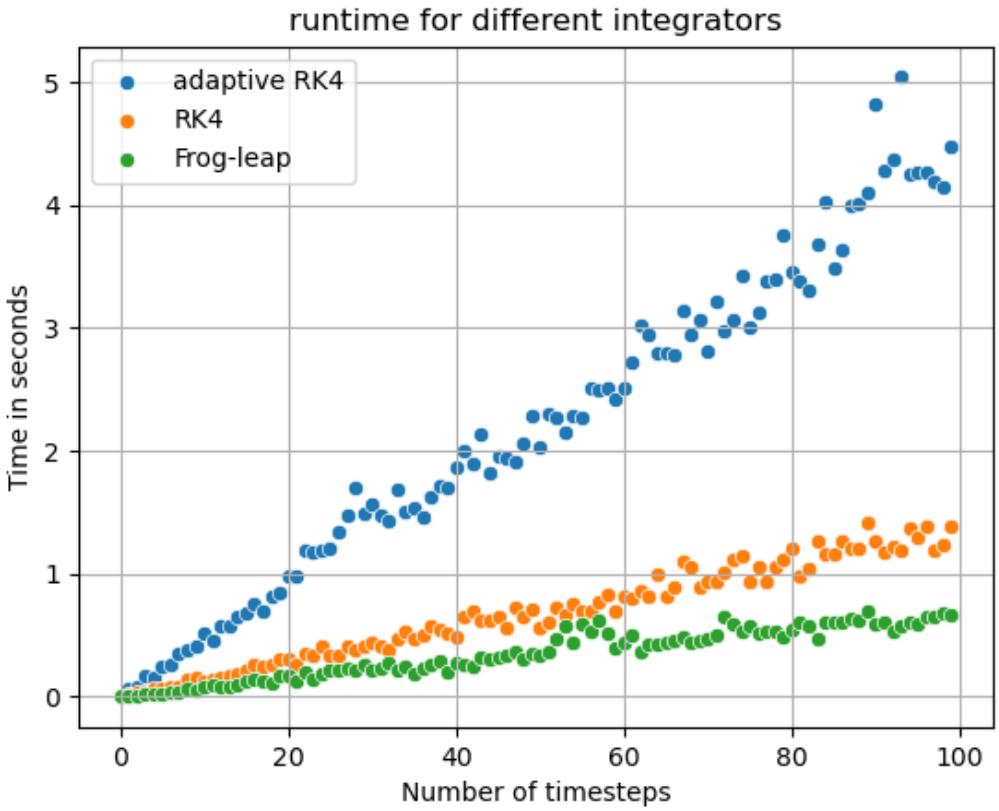


Figure 4: In this figure, we see the runtime on the y-axis and the number of timesteps used on the x-axis. In blue we see the RK4-integrator using adaptive timesteps, in orange the regular RK4 and finally in green the frog-leap integrator. For this figure I used $\theta = 0$ and $N = 10$.

4 Improvements and shortcomings

The obvious improvement we could make would be to the code would to add another dimension to the simulation. This wouldn't be all that hard conceptually but it would require to edit every single function in the code to incorporate that third dimension. It would be tedious work. A smaller improvement could be the mass distribution. We could add a randomizer for the mass per particle. Now we set this to a constant, but it would be really cool to replicate the mass distribution of the solar system for example.

The smoothing parameter ϵ is now mainly used to make sure the force between to particles doesn't blow up, this could also be done by adding a size parameter to the particles. This would make sure the particles don't overlap. I suspect this is one of the main reasons why (sometimes) the energy conservation in my code doesn't seem to hold. This would also need to be implemented in the generation of the random particles. We could extend this idea by making the particles able to bounce (elastically) against each other as they collide. This is at the moment the biggest shortcoming of the code since it isn't consistently energy-conserving. Tweaking of the hyper-parameters (θ and ϵ) could also fix this issue but we would need to run multiple simulations using the same random seed to figure out what combination works. This of course isn't ideal for big simulations. Decreasing the time-step would also be an option but that would also increase our run-speed.

Yet another improvement could go in the form of a higher order integrator. A 6-th order Runge-Kutta could be implemented but this would decrease the run-speed as we would need to calculate the forces on the particles more times per time-step than we do now. Another integrator we could use is Beeman's integration, this method however is not self-starting and would thus be harder to implement in the code. This integrator is better in conserving energy than the frog-leap algorithm[1].

Some calculations inside the tree could also be sped up. For example, inside the QuadTree class we use the 'compute total energy' method to calculate the total energy of the entire tree. The contribution of the gravitational energy is calculated with a double for loop, This means we count each pair of particles twice. To counteract this we divide by two in the final calculation but it would speed up the calculation if we iterate over each pair once instead of twice.

Lastly, I have hardcoded some parameters that preferably should not be hardcoded. For example, the $\epsilon_{\mathbf{want}}$ in the $RK4-$adaptive integrator is automatically set to a standard value. This could easily be a free parameter but i thought it would be better to hardcode them so the code wouldn't be too overwhelming to use. Another example of this is the way we control the length and time-step of a simulation.

References

[1] Harvey Gould, Jan Tobochnik, and Wolfgang Christian. *An Introduction to Computer Simulation Methods: Applications to Physical Systems*. Addison-Wesley Professional, January 2007.