Fluid Dynamics Simulation using Parallel Haskell

Joheen Chakraborty, Elyas Obbad
{jc5110, eo2446}@columbia.edu
https://github.com/joheenc/haskell-3D-MD

December 21, 2021

1 Introduction

We used Haskell to parallelize a classic problem in computational physics: simulating the molecular dynamics of liquid argon in a closed container. Given a box of side length L containing N liquid argon particles, the system can be well-approximated to behave as a classical fluid interacting under the Lennard-Jones potential. The de-dimensionalized Lennard-Jones potential between two point-mass particles is given by:

$$U(r) = 4\left[\left(\frac{1}{r}\right)^{12} - \frac{1}{2}\left(\frac{1}{r}\right)^{6}\right]$$

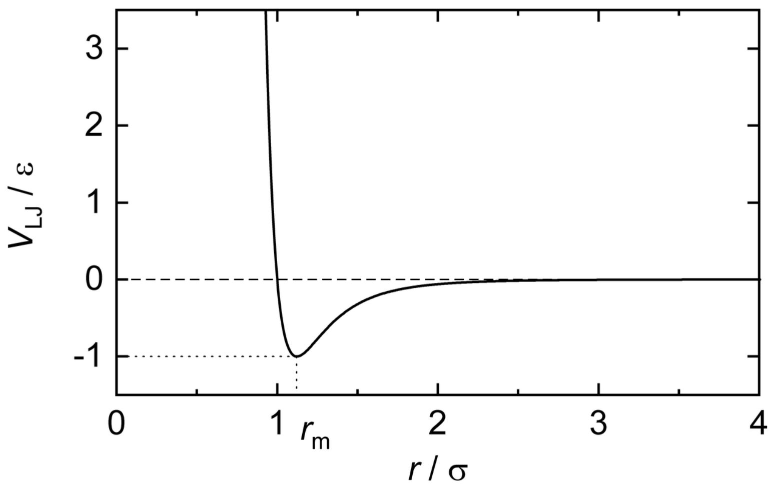


Figure 1: Illustration of the Lennard-Jones potential as a function of distance between two particles. At far distances, the potential results in an attractive force, while at close distances, the repulsive component dominates.

Our simulation will be composed of discrete time steps of duration τ. At each time step, we will have 3-dimensional vectors representing the positions $(\vec{r}_1, \vec{r}_2, \ldots, \vec{r}_N)$, velocities $(\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_N)$, and net forces $(\vec{F}_1, \vec{F}_2, \ldots, \vec{F}_N)$ of each particle. The net force on each particle can be calculated from the potential as:

$$\vec{F}_i = -\sum_{j \neq i}^{N} (\vec{r_j} - \vec{r_i}) \frac{dU}{dr_{ij}} = \sum_{j \neq i}^{N} (\vec{r_i} - \vec{r_j}) \left[\left(\frac{1}{r_{ij}}\right)^{14} - \frac{1}{2}\left(\frac{1}{r_{ij}}\right)^{8}\right]$$

Then, we can approximate the updated positions and velocities of each particle in the simulation via the following recurrence relations:

$$\vec{r}_{n+1} = \vec{r}_n + \vec{v}_n\tau + \left(\tau^2/2\right)\vec{F}_n \tag{1}$$

$$\vec{v}_{n+1/2} = \vec{v}_n + \left(\tau/2\right)\vec{F}_n \tag{2}$$

$$\vec{v}_{n+1} = \vec{v}_{n+1/2} + \left(\tau/2\right)\vec{F}_{n+1} \tag{3}$$

where n indexes the time steps. Any starting configuration can work, but a reasonable baseline is to begin the simulation with equally spaced, static particles.

This simulation lends itself naturally to data parallelism, which we can accomplish using Strategies in Haskell. In Section 2, we describe our approach to implementing the physics simulator in Haskell, and our method for visualizing the fluid dynamics using graphics libraries. In Section 3, we describe our parallelization strategy, as well as examine the computational performance gained for various test cases. In Section 4 we make concluding remarks and suggest more parallelization schemes to further optimize the simulation, and in Section 5 we list the code written for the project.

2 Implementation

2.1 Physics simulation

The core of our simulation is the `Atom` data type, which defines a single atom of liquid argon for our simulation:

```haskell
import Linear.V3
data Atom = Atom { i :: Int,
                   r :: V3 Float,
                   v :: V3 Float }
```

`i` represents the index of each `Atom` in the list; this is used to ensure forces are not calculated between the same atom. `r` and `v` are vectors of floats represent the components of position and velocity, respectively, which are stored using vectors from the `Linear` module for easily performing linear algebraic operations.

Our fluid model is then represented as a `[Atom]` with many constituent fluid particles interacting under the Lennard-Jones potential. Simulation updates are made according to Equations (1), (2), and (3). Recall Equation (1), which describes the position update step, is:

$$\vec{r}_{n+1} = \vec{r}_n + \vec{v}_n\tau + \left(\tau^2/2\right)\vec{F}_n$$

This equation is implented as:

```haskell
rstep :: Float -> Atom -> V3 Float -> Atom
rstep dt (Atom i r v) a = Atom i r' v
  where r' = r ^+^ (v ^* dt) ^+^ (0.5*dt**2 *^ a)
```

The ˆ+ˆ operator is specific to the `Linear` module, and it represents vector addition. Likewise, the module also provides similar operators for vector subtraction (ˆ−ˆ) and multiplication of a vector by a scalar (ˆ* and *ˆ).

The velocity update steps as given in Equations (2) and (3) are:

$$\vec{v}_{n+1/2} = \vec{v}_n + \left(\tau/2\right)\vec{F}_n$$

$$\vec{v}_{n+1} = \vec{v}_{n+1/2} + \left(\tau/2\right)\vec{F}_{n+1}$$

The code implementation of the velocity update step is as follows:

```haskell
vstep :: Float -> Atom -> V3 Float -> Atom
vstep dt atom a = Atom i r v'
  where (Atom i r v) = atom
```

```
        v' = (bound atom) * (v + (0.5 * dt) *^ a)
        bound (Atom _ (V3 x y z) _) = V3 xf yf zf
          where xf = if (abs x + rad > s/2) then (-1) else 1
                yf = if (abs y + rad > s/2) then (-1) else 1
                zf = if (abs z + rad > s/2) then (-1) else 1
```

As we see above, the velocity Verlet algorithm—our approach to the simulation—is a "leap-frog" algorithm, i.e. there are two velocity updates for each position update. This allows a more accurate correspondence of accelerations to velocities, increasing the granularity of time-steps with which we update particle trajectories. The notable difference with the original velocity Verlet algorithm equation is due to our boundary conditions, which require that particles will bounce back once they reach the edge of the container. This is done by reversing the sign on the relevant component of its velocity.

Forces are then calculated via the Lennard-Jones potential as given in Section 1:

```
fstep :: [Atom] -> [V3 Float]
fstep atoms = fTot atoms atoms
  where fTot [a] bs = fOne a bs
        fTot (a:as) bs = fOne a bs ^+^ fTot as bs

{- helper function to calculate the total net force acting on a single atom -}
fOne :: Atom -> [Atom] -> [V3 Float]
fOne atom = fmap (f atom)
  where f a b = if (i a==i b) then 0 else lennardJones a b
        lennardJones a b = (1 / (norm d)^14 - 0.5 / (norm d)^8) *^ d
          where d = (r b) ^-^ (r a)
```

So, we can finally combine these separate update functions to define a single "step" function for our algorithm:

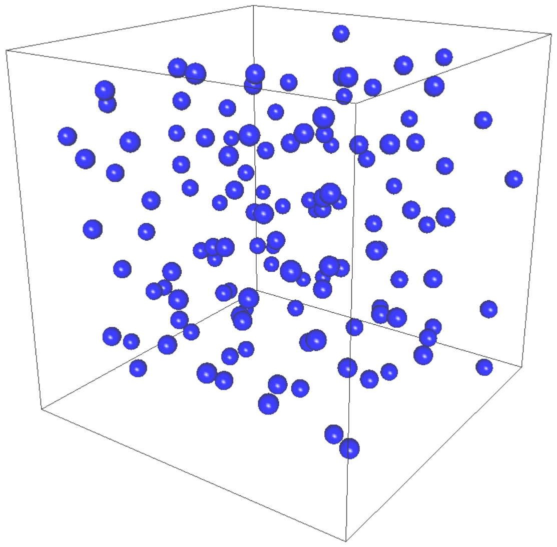
```
step :: Float -> [Atom] -> [Atom]
step dt atoms = zipWith (vstep dt) r' f'
  where f = fstep atoms
        r' = zipWith (rstep dt) atoms f
        f' = f ^+^ (fstep r')
```

Examining this code gives us an intuition for the physical "work" being done by the algorithm: at each timestep, the net force on each atom by all the others is calculated and stored in `f`. Because we have de-dimensionalized the equations of motion, such that masses are in units of argon atomic mass, the forces and accelerations are equivalent; so, the updated position vectors, `r'`, are then computed directly from these forces and the velocities of each `Atom`. The second update of force, `f'`, accounts for the leap-frog step as described with the `vstep` implementation above. Finally, we update $\vec{v}_{n+1}$ by calling `vstep` along with `r'` and `f'`.

2.2 Animation

We used the `not-gloss` library (https://hackage.haskell.org/package/not-gloss) to animate our fluid dynamics simulator at work. `not-gloss` provides a convenient Haskell interface to OpenGL, allowing us to create an on-screen window and use our `step` function to update 50 frames per second as an animation of the particle motion. There are a variety of other 3D graphics libraries suitable for our simulation, however lots of them lacked clear documentation or readable examples. While `not-gloss` does not have extensive documentation, we found it more intuitive than other 3D graphics libraries due to its similarity to `gloss`, a popular and well-documented 2D vector graphics, animations, and simulations library.

Here is a screenshot of the 3D animation in action, for a sample simulation of 125 particles:



A GIF of the animation can be viewed on the GitHub page of our project, https://github.com/joheenc/haskell-3D-MD.

3 Parallelization

We thought carefully about how to efficiently parallelize our algorithm, which is comprised of several steps of varying runtimes. For a simulation of n atoms:

- The position update step is $O(n)$, as we simply iterate through each atom to move it by its corresponding velocity
- The velocity update step is also $O(n)$, as we perform this linear iteration of updates by the force twice (once for each half-step in the leap-frog algorithm)
- The force update step is $O(n^2)$. Because the net force on each particle depends on its distances from every other particle in the simulation, we require a quadratic-time iteration to calculate total force/acceleration on each atom in the list.

Thus, for t time-steps, the algorithm runs in total $O(tn^2)$ time. Given that the force update is most expensive, our first approach was to directly parallelize it using the `par` function of `Control.Parallel`.

```
import Control.Parallel(par)
fstep :: [Atom] -> [V3 Float]
fstep atoms = fTot atoms atoms
  where fTot [x] ys = fOne x ys
        fTot (x:xs) ys = par fPar (fPar ^+^ fTot xs ys)
          where fPar = fOne x ys
```

However, we found this primitive approach to be of little value. The comptuational overhead from parallelizing the code completely nullified the speedup from running the force update in parallel, and in most cases this code performed worse than the single-threaded version.

The next approach we turned to was using the `Control.Parallel.Strategies` module. This module has several convenient built-in strategies to partition lists and handle computation in parallel, such as `parMap`, `parList`, `parListChunk`, and `parBuffer`. The Strategies most directly suited to our task at hand are `parList` and `parListChunk`, since they are explicitly designed for parallel list computation. Using these functions, we could divide the n atoms in our list into smaller sub-lists, and perform all physics updates across the sub-lists in parallel. Either `parList` or `parListChunk` could be used to this effect, but we found the latter to perform slightly better, assuming we designated the chunk size to be the total number of atoms divided by the number of cores used. In this case, the parallelization becomes like so:

```haskell
import Control.Parallel.Strategies
step :: Float -> [Atom] -> [Atom]
step dt atoms = zipWith (vstep dt) r' f'
  where f = fstep atoms `using` parListChunk chunkSize rdeepseq
        r' = zipWith (rstep dt) atoms f
        f' = f ^+^ (fstep r' `using` parListChunk chunkSize rdeepseq)
```

Where the `chunkSize` for `parListChunk` is computed as described above. Note that we did not parallelize the `r'` or `v'` update steps, though both of those could just as easily be done with the same functions: in our tests, we found that parallelizing these steps results in no speedup (in some cases, the performance actually decreases), because the additional overhead is not successfully offset. We used `rdeeseq` to completely evaluate each of the `Atoms` to normal form.

Of the strategies we tested, the chunking strategy performs best, likely because it creates less sparks and thus suffers less overhead in parallelization. For details on the performance boosts for N=1 to 4 cores with various physical parameters, see Figures 1-3. The notable issue we faced, depicted in Figure 4, is that in spite of the parallel CPUs running through most of the simulation duration, the total CPU activity at any given time does not nearly approach the sum of the four CPUs. Thus, there is likely a limitation of our parallelization scheme that may be handled via other approaches, e.g. using the `Repa` module to handle data parallelism.

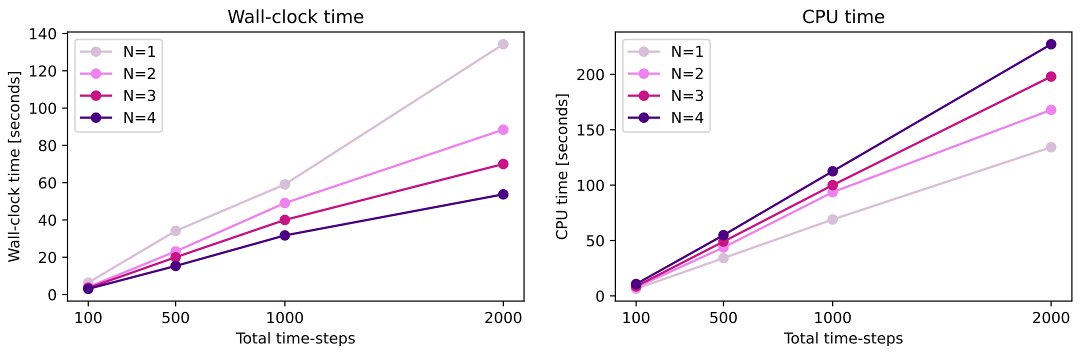


Figure 2: Runtimes for several simulations of 256 atoms and increasing simulation lengths, ranging from 100 to 2000 time-steps. The parallelization provides a performance boost up to N=4 cores.

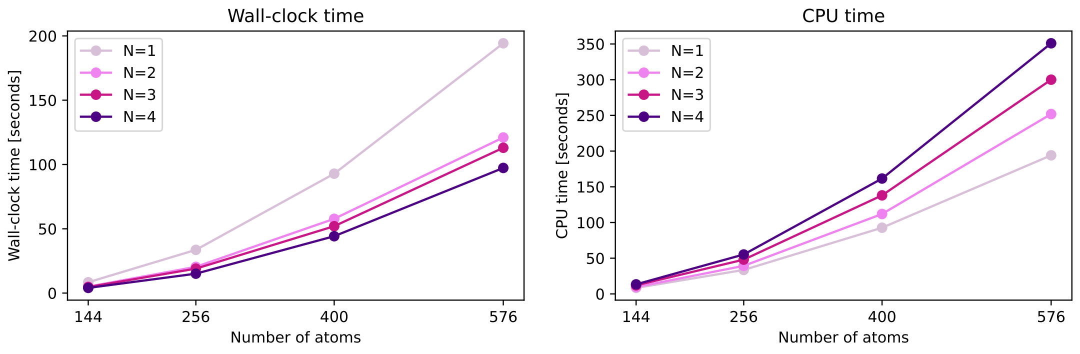


Figure 3: Runtimes for several simulations of 500 time steps and increasing atom numbers, ranging from 144 to 576 atoms. We see a similar trend as above, with N=4 cores providing the best speedup. For each simulation, we use `parListChunk` with a chunk size chosen to equally partition the list across the number of cores.

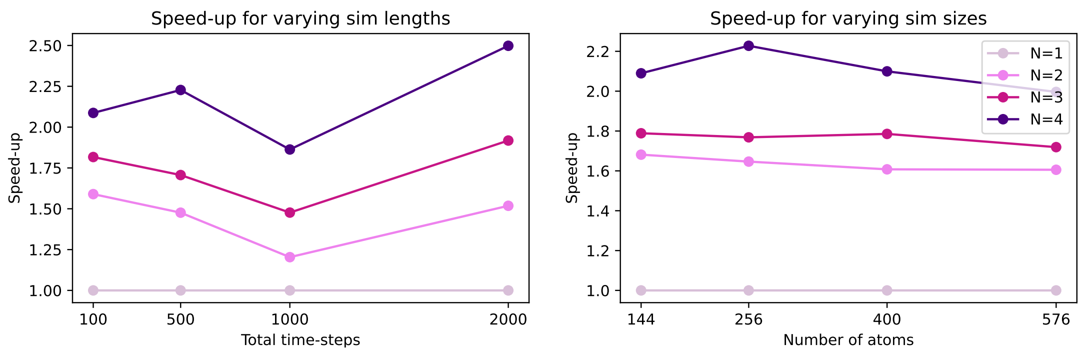


Figure 4: Relative speedups from the simulations of varying length (left) and size (right). The relative speedup peaks around 2.5x, which we can achieve using N=4 cores, but adding additional cores contributes too much computational overhead to be worth it in the end.

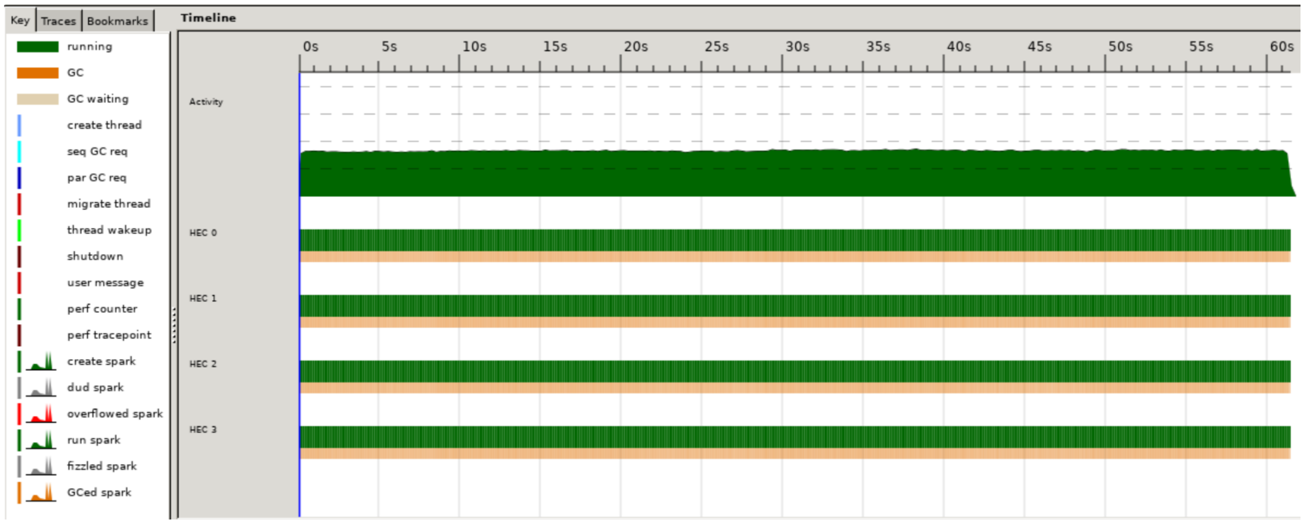


Figure 5: Threadscope CPU usage for the 2000 time-step, 256-particle case across 4 cores (speedup of 2.5x). Although all four CPU cores are continuously working throughout the simulation, the total activity at any given time is much lower than 4x, which likely explains why the speedup plateaus at only 2.5x in spite of using 4 cores. We find from zooming into the Threadscope profile that this is related to frequent garbage collection as a result of the steep memory costs of the physics simulator. However, we couldn't find an easy way around this, leaving us at a 2.5x speedup for now.

As a final addition, we also parallelized the animation rendering step of the `not-gloss` library like so:

```
draw config = VisObjects $ [box] ++ (drawAtom <$> config `using` parListChunk chunkSize rseq)
  where box = Trans (V3 0 0 0) $ Box (s, s, s) Wireframe black
            drawAtom atom = Trans (r atom) $ Sphere rad Solid blue
```

This way, the atoms themselves are rendered in parallel, allowing for more computationally efficient animation. As an example of the benefit of parallelizing the drawing function, we tested the `mainAnim` function with a 5x5x5 grid of particles, which runs smoothly for N=4 cores, but is unable to animate without serious lag for N=1 core.

4 Conclusion

We found that parallelization in Haskell, while very convenient to implement, takes careful consideration and cost-benefit analysis of the computational overhead that comes with designated workloads to carry out in parallel. In the end, we found that our best speedups actually came from selectively parallelizing only the quadratic-time segments of our simulator, leaving the linear-time parts of our algorithm to run on a single thread. In the best case, we were able to achieve a speedup of 2.5x using 4 cores; this result was hindered by excessive garbage collection, which is a difficult problem to circumvent using Haskell lists. Future work could thus experiment with, e.g., using `Vectors` instead of lists to store data, or using the `Repa` module, which also is well-suited for handling arrays in parallel.

5 Code listing

Also available at https://github.com/joheenc/haskell-3D-MD

```haskell
import Linear.Metric
import Linear.V3
import Linear.Vector
import Control.Parallel.Strategies
import Vis

n = 4 {- grid size, i.e. initializes n x n x n cube of atoms
         (don't use n>5 for animated executions; for n=5, you will likely need to use N=4 cores)-}
chunkSize = 32 {- chunk size to use in parListChunk. Set this to numCores / n^3 -}
s = 7 {- cubical container side length; if you have a large n, e.g. >=9, make sure to expand the box
    so there is space
        for the particles to take discrete time steps without getting unphysically close together,
            e.g. s >= 10 -}
simLen = 300 {- number of time steps to run the simulation for, if you're using mainNoAnim -}
main :: IO ()
main = mainAnim {-choose from mainAnim or mainNoAnim to either run the
                  simulation 3D animated in a GUI window, or run a finite number
                  of time steps without any animation-}

{- more parameters; you probably don't want to change these -}
rad = 0.15 {- atom radius -}
dt = 0.1 {- time step length -}

{- define the Atom data type, the basic unit of our simulation -}
data Atom = Atom { i :: Int,    -- index in the array
                   r :: V3 Float, -- position vector
                   v :: V3 Float } -- velocity vector

instance Show Atom where
  show (Atom i r v) = show i ++ ": r=" ++ show r ++ ", v=" ++ show v

{- position update in linear time -}
rstep :: Float -> Atom -> V3 Float -> Atom
rstep dt (Atom i r v) a = Atom i r' v
  where r' = r ^+^ (v ^* dt) ^+^ (0.5*dt**2 *^ a)

{- velocity update in linear time -}
vstep :: Float -> Atom -> V3 Float -> Atom
vstep dt atom a = Atom i r v'
  where (Atom i r v) = atom
        v' = (bound atom) * (v + (0.5 * dt) *^ a)
        bound (Atom _ (V3 x y z) _) = V3 xf yf zf -- enforces rigid wall boundary condition
```

```haskell
          where xf = if (abs x + rad > s/2) then (-1) else 1
                yf = if (abs y + rad > s/2) then (-1) else 1
                zf = if (abs z + rad > s/2) then (-1) else 1

{- force update in quadratic time -}
fstep :: [Atom] -> [V3 Float]
fstep atoms = fTot atoms atoms
  where fTot [a] bs = fOne a bs
        fTot (a:as) bs = fOne a bs ^+^ fTot as bs

{- helper function to calculate the total net force acting on a single atom -}
fOne :: Atom -> [Atom] -> [V3 Float]
fOne atom = fmap (f atom)
  where f a b = if (i a==i b) then 0 else lennardJones a b
        lennardJones a b = (1 / (norm d)^14 - 0.5 / (norm d)^8) *^ d
          where d = (r b) ^-^ (r a)

{- velocity Verlet algorithm -}
step :: Float -> [Atom] -> [Atom]
step dt atoms = zipWith (vstep dt) r' f'
  where f = fstep atoms `using` parListChunk chunkSize rdeepseq
        r' = zipWith (rstep dt) atoms f
        f' = f ^+^ (fstep r' `using` parListChunk chunkSize rdeepseq)

{- run the program with animation enabled -}
mainAnim = simulate options refreshRate initConfig draw update
  where options =
          ( defaultOpts
            { optWindowName = "FluidDyn",
              optBackgroundColor = Just white,
              optWindowSize = Just (1280, 720)
            }
          )
        refreshRate = 0.02
        initConfig = grid n
        draw config = VisObjects $ [box] ++ (drawAtom <$> config `using` parListChunk chunkSize rseq)
          where box = Trans (V3 0 0 0) $ Box (s, s, s) Wireframe black
                drawAtom atom = Trans (r atom) $ Sphere rad Solid blue
        update _ config = step dt config

{- run the program with no animation -}
mainNoAnim = runSim simLen (grid n)
  where runSim :: Int -> [Atom] -> IO ()
        runSim 0 model' = do
          putStrLn (show $ head model')
          return ()
        runSim n model = do
          let model' = step dt model
          runSim (n-1) model'

{- initialize a n x n x n cubical grid as the initial atom configuration -}
grid :: Int -> [Atom]
grid n = zipWith3 Atom [1..(n^3)] (cube n n) (replicate (n^3) (V3 0 0 0))
  where cube _ 0 = []
        cube d i = square d d z ++ cube d (i-1)
          where z = s/2 - (fromIntegral i * s/fromIntegral (d+1))

square :: Int -> Int -> Float -> [V3 Float]
square _ 0 _ = []
square d i z = row d d y ++ square d (i-1) z
```

```
 where y = s/2 - (fromIntegral i * s/fromIntegral (d+1))
       row _ 0 _ = []
       row d i y = V3 x y z : row d (i-1) y
         where x = s/2 - (fromIntegral i * s/fromIntegral (d+1))
```