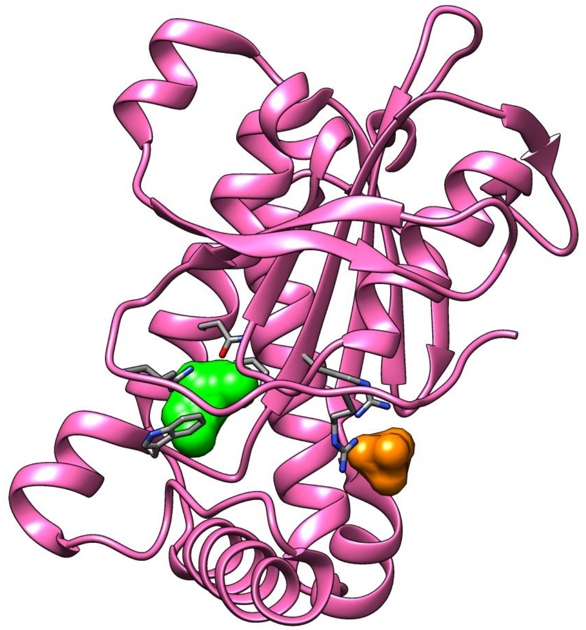


Julia: a natural language for computational biology

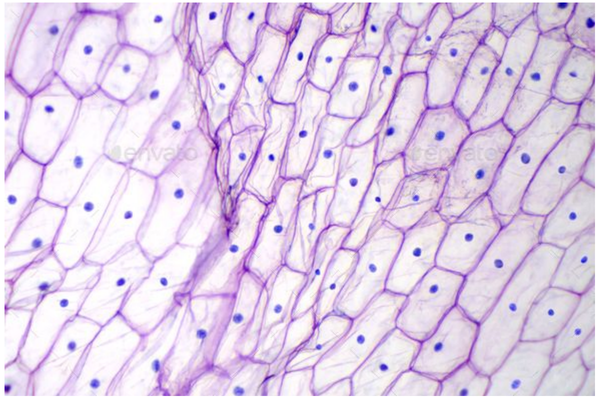
Joe Greener
Research Associate, UCL
jgreener64.github.io

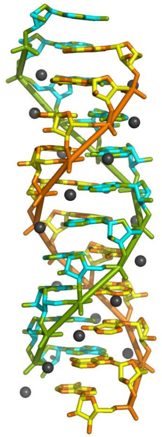
Talk outline

- Computational biology
- Julia
- BioJulia
- BioStructures.jl
- Bio3DView.jl
- Molly.jl
    - Simulating an ideal gas
    - Simulating diatomic molecules
    - Simulating proteins
- Differentiable molecular simulation
- Assessing Julia
- The future

What is biology?

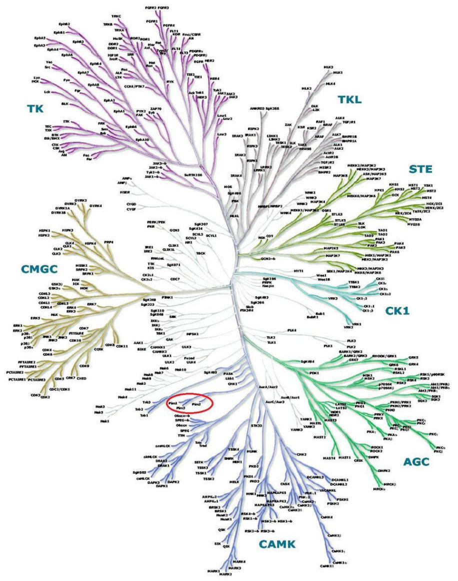
“Biology is the natural science that studies life and living organisms, including their physical structure, chemical processes, molecular interactions, physiological mechanisms, development and evolution.” - Wikipedia





What *is* biology?

From Manning et al, *Science*, 2002



- Complex systems
- Multi-scale systems
- Lots of data
- Incomplete, noisy data
- Rules have exceptions

Computational biology addresses this by developing and applying data analysis, mathematical modeling and simulation techniques to biology.

Protein structure prediction

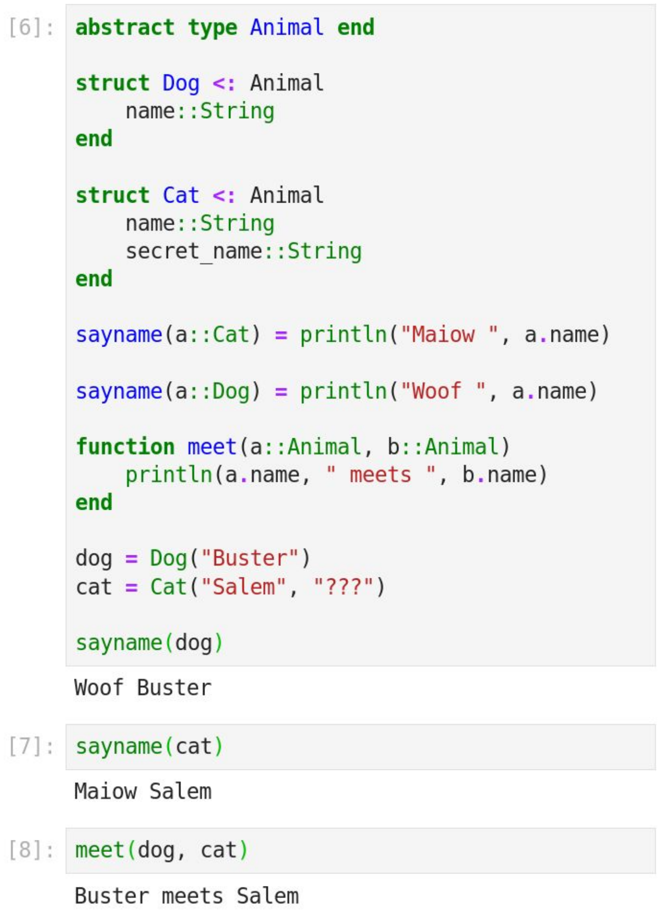
- My day job involves predicting the structure of proteins from the sequence of amino acids that makes them up
- We do this using deep learning (CNNs/ResNets) on evolutionary information
- https://github.com/psipred/DMPfold

```
MENFQKVEKIGEGTYGVVYKARNKLTGEVVALKKIRLDTETEGVPSTAIREISLLKE
LNHPNIVKLLDVIHTENKLYLVFEFLHQDLKKFMDASALTGIPLPLIKSYLFQLLQG
LAFCHSHRVLHRDLKPQNLLINTEGAIKLADFGLARAFGVPVRTYTHEVVTLWYRAP
EILLGCKYYSTAVDIWSLGCIFAEMVTRRALFPGDSEIDQLFRIFRTLGTPDEVVWP
GVTSMPDYKPSFPKWARQDFSKVVPPLDEDGRSLLSQMLHYDPNKRISAKAALAHPF
FQDVTKPVPHLRL
```

$\longrightarrow$

Julia

- High-level programming language
- Fast → addresses the two-language problem
- Free, open source
- Strong community
- Dynamically typed
- Multiple dispatch

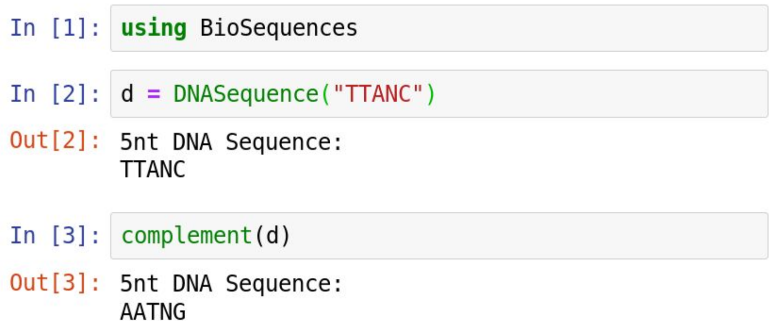


```
[6]: abstract type Animal end

     struct Dog <: Animal
         name::String
     end

     struct Cat <: Animal
         name::String
         secret_name::String
     end

     sayname(a::Cat) = println("Maiow ", a.name)

     sayname(a::Dog) = println("Woof ", a.name)

     function meet(a::Animal, b::Animal)
         println(a.name, " meets ", b.name)
     end

     dog = Dog("Buster")
     cat = Cat("Salem", "???")

     sayname(dog)

     Woof Buster

[7]: sayname(cat)

     Maiow Salem

[8]: meet(dog, cat)

     Buster meets Salem
```

BioJulia

- Fast, open, easy, software for biology
- The bioinformatics infrastructure for the Julia language

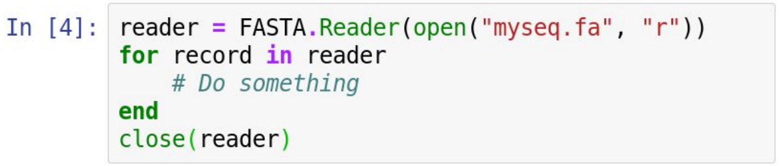
Efficient sequence type



```
In [1]: using BioSequences

In [2]: d = DNASequence("TTANC")
Out[2]: 5nt DNA Sequence:
        TTANC

In [3]: complement(d)
Out[3]: 5nt DNA Sequence:
        AATNG
```

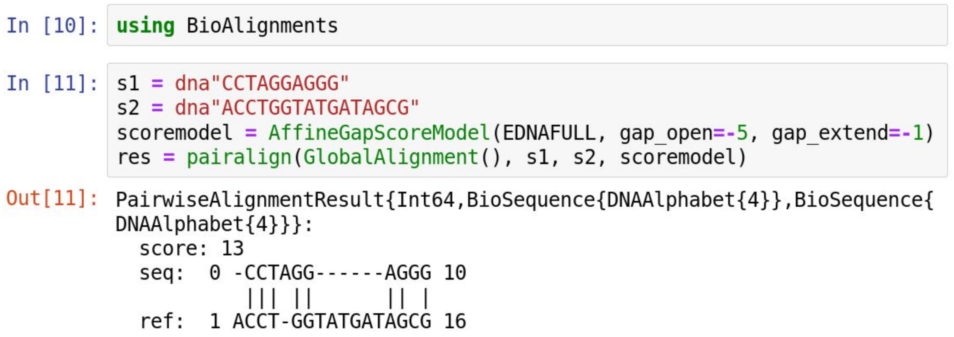
BioJulia

File parsers



```
In [4]: reader = FASTA.Reader(open("myseq.fa", "r"))
        for record in reader
            # Do something
        end
        close(reader)
```

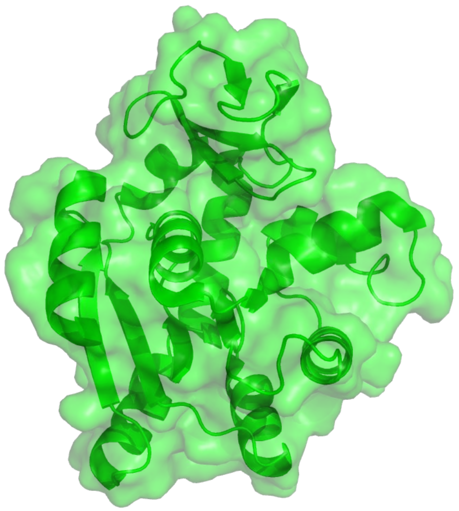
Alignments



```
In [10]: using BioAlignments

In [11]: s1 = dna"CCTAGGAGGG"
         s2 = dna"ACCTGGTATGATAGCG"
         scoremodel = AffineGapScoreModel(EDNAFULL, gap_open=-5, gap_extend=-1)
         res = pairalign(GlobalAlignment(), s1, s2, scoremodel)

Out[11]: PairwiseAlignmentResult{Int64,BioSequence{DNAAlphabet{4}},BioSequence{
         DNAAlphabet{4}}}:
           score: 13
           seq:  0 -CCTAGG------AGGG 10
                    ||| ||      || |
           ref:  1 ACCT-GGTATGATAGCG 16
```

BioStructures.jl

- Read, write and manipulate macromolecular structures
- https://github.com/BioJulia/BioStructures.jl



PDB format

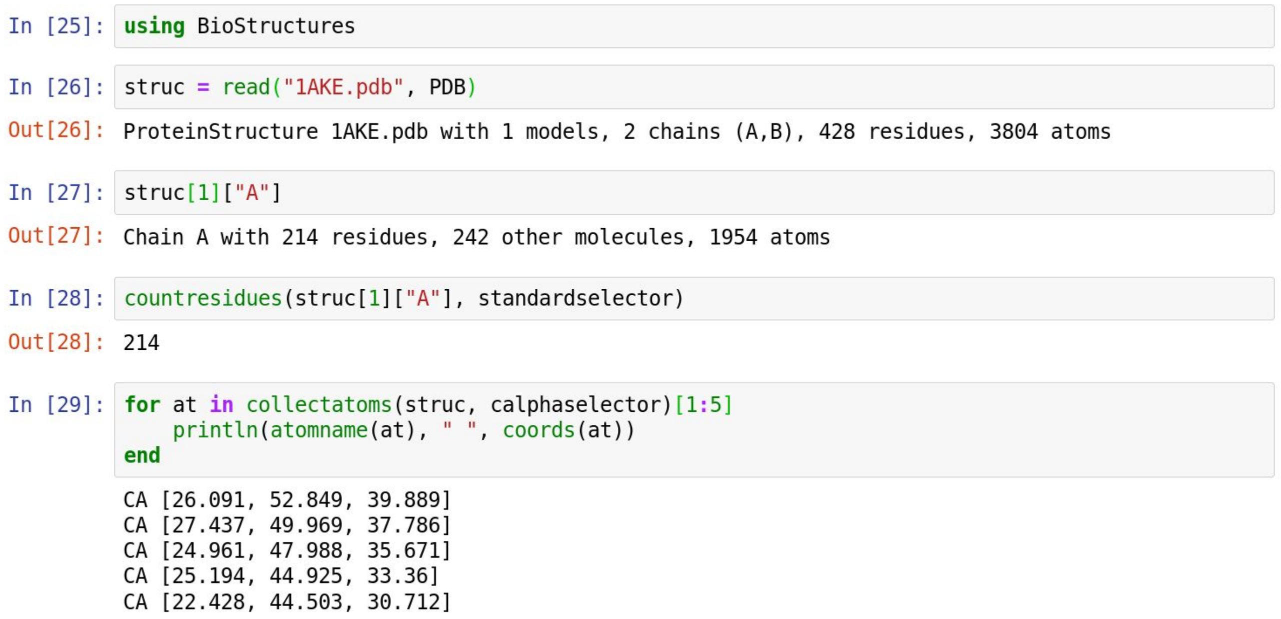
```
ATOM      1  N   MET A   1      26.981  53.977  40.085  1.00 40.83           N
ATOM      2  CA  MET A   1      26.091  52.849  39.889  1.00 37.14           C
ATOM      3  C   MET A   1      26.679  52.163  38.675  1.00 30.15           C
ATOM      4  O   MET A   1      27.020  52.865  37.715  1.00 27.59           O
ATOM      5  CB  MET A   1      24.677  53.310  39.580  1.00 38.06           C
ATOM      6  CG  MET A   1      23.624  52.189  39.442  1.00 46.67           C
ATOM      7  SD  MET A   1      21.917  52.816  39.301  1.00 61.54           S
ATOM      8  CE  MET A   1      21.930  53.926  37.910  1.00 51.17           C
ATOM      9  N   ARG A   2      26.861  50.841  38.803  1.00 28.23           N
ATOM     10  CA  ARG A   2      27.437  49.969  37.786  1.00 25.76           C
```

mmCIF format

```
ATOM   1    N N   . MET A 1 1   ? 26.981 53.977  40.085 1.00 40.83  ? 1   MET A N   1
ATOM   2    C CA  . MET A 1 1   ? 26.091 52.849  39.889 1.00 37.14  ? 1   MET A CA  1
ATOM   3    C C   . MET A 1 1   ? 26.679 52.163  38.675 1.00 30.15  ? 1   MET A C   1
ATOM   4    O O   . MET A 1 1   ? 27.020 52.865  37.715 1.00 27.59  ? 1   MET A O   1
ATOM   5    C CB  . MET A 1 1   ? 24.677 53.310  39.580 1.00 38.06  ? 1   MET A CB  1
ATOM   6    C CG  . MET A 1 1   ? 23.624 52.189  39.442 1.00 46.67  ? 1   MET A CG  1
ATOM   7    S SD  . MET A 1 1   ? 21.917 52.816  39.301 1.00 61.54  ? 1   MET A SD  1
ATOM   8    C CE  . MET A 1 1   ? 21.930 53.926  37.910 1.00 51.17  ? 1   MET A CE  1
ATOM   9    N N   . ARG A 1 2   ? 26.861 50.841  38.803 1.00 28.23  ? 2   ARG A N   1
ATOM   10   C CA  . ARG A 1 2   ? 27.437 49.969  37.786 1.00 25.76  ? 2   ARG A CA  1
```

BioStructures.jl

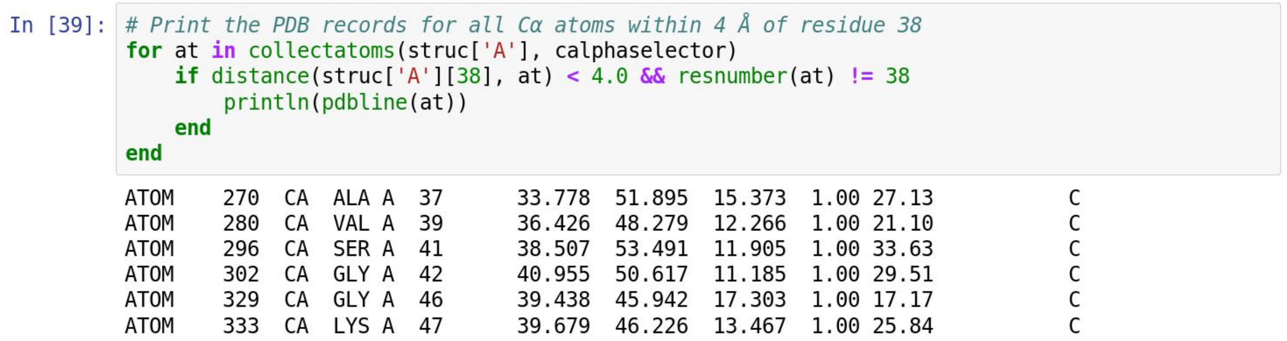
- Read, write and manipulate macromolecular structures



```
In [25]: using BioStructures

In [26]: struc = read("1AKE.pdb", PDB)

Out[26]: ProteinStructure 1AKE.pdb with 1 models, 2 chains (A,B), 428 residues, 3804 atoms

In [27]: struc[1]["A"]

Out[27]: Chain A with 214 residues, 242 other molecules, 1954 atoms

In [28]: countresidues(struc[1]["A"], standardselector)

Out[28]: 214

In [29]: for at in collectatoms(struc, calphaselector)[1:5]
             println(atomname(at), " ", coords(at))
         end

         CA [26.091, 52.849, 39.889]
         CA [27.437, 49.969, 37.786]
         CA [24.961, 47.988, 35.671]
         CA [25.194, 44.925, 33.36]
         CA [22.428, 44.503, 30.712]
```

BioStructures.jl

- Spatial calculations



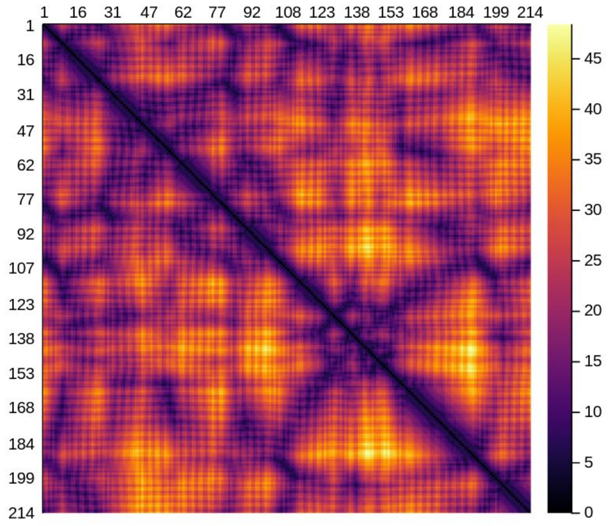
```
In [39]: # Print the PDB records for all Cα atoms within 4 Å of residue 38
         for at in collectatoms(struc['A'], calphaselector)
             if distance(struc['A'][38], at) < 4.0 && resnumber(at) != 38
                 println(pdbline(at))
             end
         end
```

```
ATOM    270  CA  ALA A  37      33.778  51.895  15.373  1.00 27.13           C
ATOM    280  CA  VAL A  39      36.426  48.279  12.266  1.00 21.10           C
ATOM    296  CA  SER A  41      38.507  53.491  11.905  1.00 33.63           C
ATOM    302  CA  GLY A  42      40.955  50.617  11.185  1.00 29.51           C
ATOM    329  CA  GLY A  46      39.438  45.942  17.303  1.00 17.17           C
ATOM    333  CA  LYS A  47      39.679  46.226  13.467  1.00 25.84           C
```

BioStructures.jl

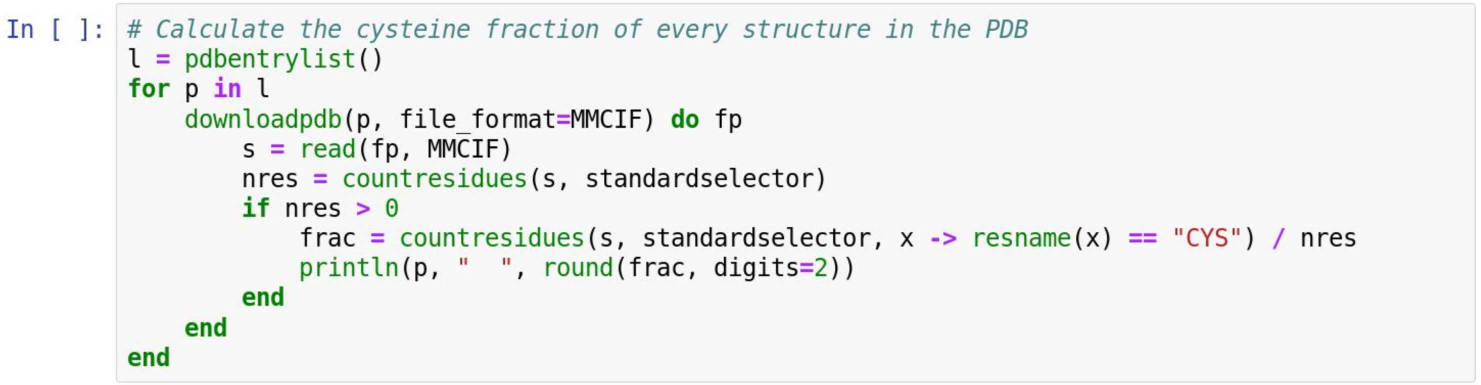
- Spatial calculations

```
In [33]: dists = DistanceMap(collectatoms(struc['A'], cbetaselector))
Out[33]: Distance map of size (214, 214)

In [34]: using Plots
         plot(dists)
Out[34]:
```



BioStructures.jl

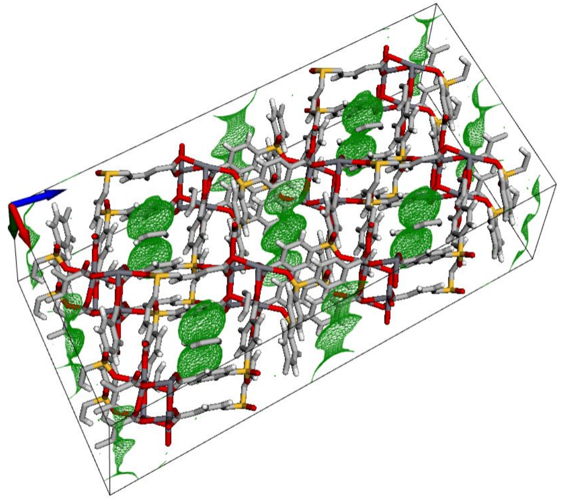
- Access the Protein Data Bank (PDB)



```
In [ ]: # Calculate the cysteine fraction of every structure in the PDB
        l = pdbentrylist()
        for p in l
            downloadpdb(p, file_format=MMCIF) do fp
                s = read(fp, MMCIF)
                nres = countresidues(s, standardselector)
                if nres > 0
                    frac = countresidues(s, standardselector, x -> resname(x) == "CYS") / nres
                    println(p, "   ", round(frac, digits=2))
                end
            end
        end
```

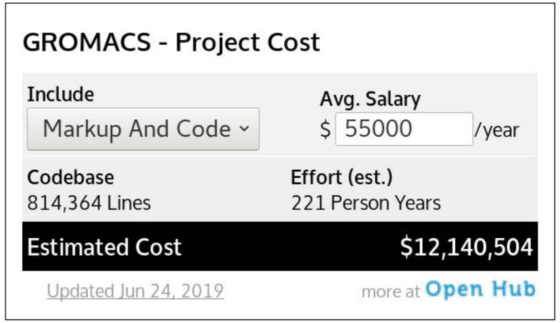
Bio3DView.jl

- Visualisation of macromolecular structures
- Wrapper round 3Dmol.js (Rego and Koes, *Bioinformatics*, 2015)
- Works in IJulia or Blink popup window
- https://github.com/jgreener64/Bio3DView.jl



Molly.jl

- Molecular dynamics (MD) is a computational technique used to explore the movement and interaction of molecules
- Molly.jl is a proof-of-concept for MD in Julia
- https://github.com/jgreener64/Molly.jl



GROMACS - Project Cost

Include: Markup And Code

Avg. Salary: $ 55000 /year

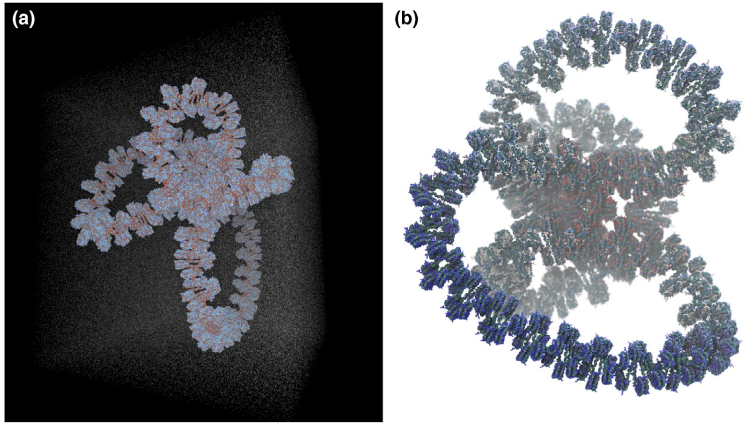
Codebase: 814,364 Lines

Effort (est.): 221 Person Years

Estimated Cost: $12,140,504

Updated Jun 24, 2019 — more at Open Hub

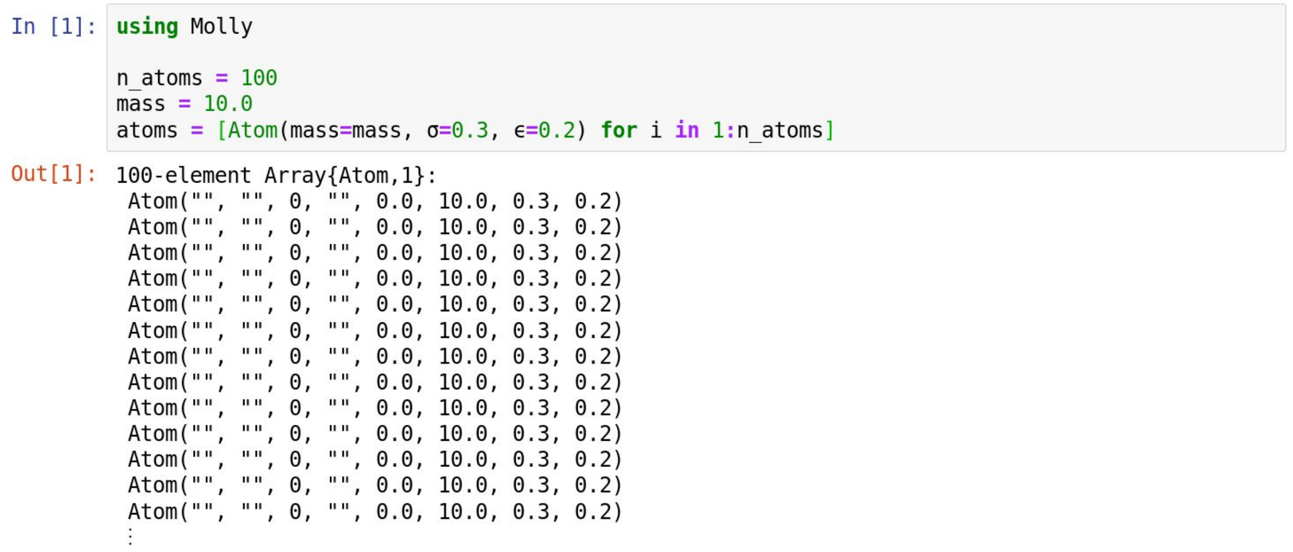
From gromacs.org



From Jung et al, *J Comp Chem*, 2019

Simulating an ideal gas

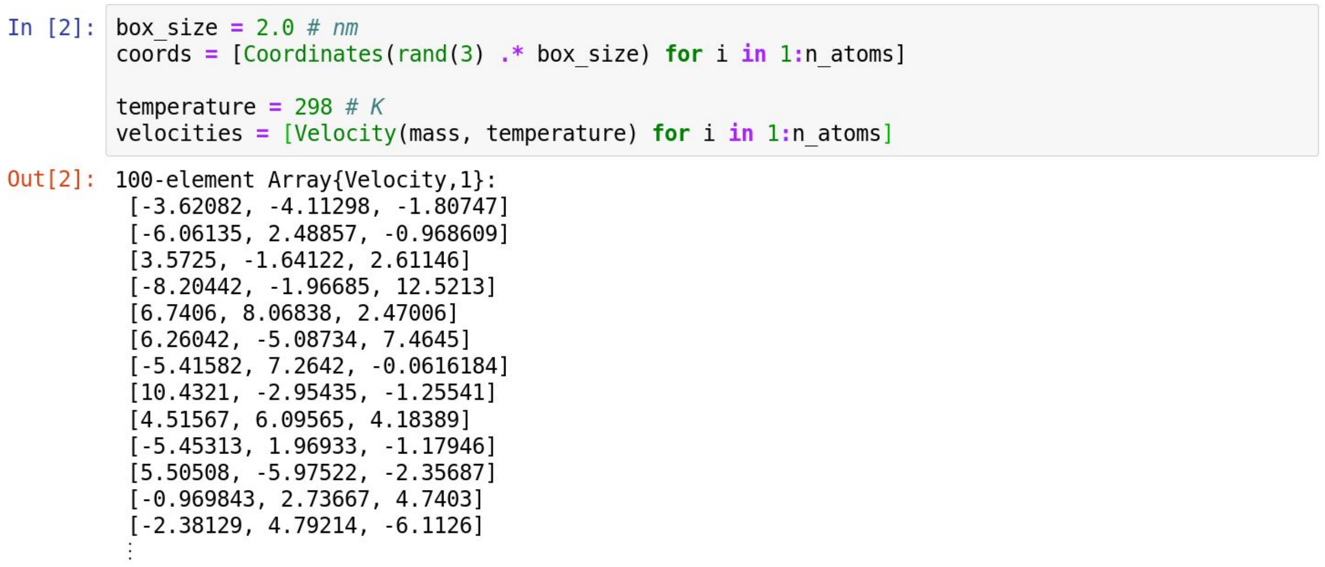
- Create some atoms with the relevant parameters defined



```
In [1]: using Molly

        n_atoms = 100
        mass = 10.0
        atoms = [Atom(mass=mass, σ=0.3, ϵ=0.2) for i in 1:n_atoms]

Out[1]: 100-element Array{Atom,1}:
         Atom("", "", 0, "", 0.0, 10.0, 0.3, 0.2)
         Atom("", "", 0, "", 0.0, 10.0, 0.3, 0.2)
         Atom("", "", 0, "", 0.0, 10.0, 0.3, 0.2)
         Atom("", "", 0, "", 0.0, 10.0, 0.3, 0.2)
         Atom("", "", 0, "", 0.0, 10.0, 0.3, 0.2)
         Atom("", "", 0, "", 0.0, 10.0, 0.3, 0.2)
         Atom("", "", 0, "", 0.0, 10.0, 0.3, 0.2)
         Atom("", "", 0, "", 0.0, 10.0, 0.3, 0.2)
         Atom("", "", 0, "", 0.0, 10.0, 0.3, 0.2)
         Atom("", "", 0, "", 0.0, 10.0, 0.3, 0.2)
         Atom("", "", 0, "", 0.0, 10.0, 0.3, 0.2)
         Atom("", "", 0, "", 0.0, 10.0, 0.3, 0.2)
         Atom("", "", 0, "", 0.0, 10.0, 0.3, 0.2)
         ⋮
```

Simulating an ideal gas

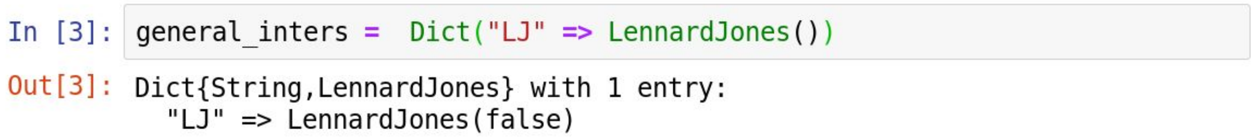
- Define some starting coordinates and velocities



```
In [2]: box_size = 2.0 # nm
        coords = [Coordinates(rand(3) .* box_size) for i in 1:n_atoms]

        temperature = 298 # K
        velocities = [Velocity(mass, temperature) for i in 1:n_atoms]

Out[2]: 100-element Array{Velocity,1}:
         [-3.62082, -4.11298, -1.80747]
         [-6.06135, 2.48857, -0.968609]
         [3.5725, -1.64122, 2.61146]
         [-8.20442, -1.96685, 12.5213]
         [6.7406, 8.06838, 2.47006]
         [6.26042, -5.08734, 7.4645]
         [-5.41582, 7.2642, -0.0616184]
         [10.4321, -2.95435, -1.25541]
         [4.51567, 6.09565, 4.18389]
         [-5.45313, 1.96933, -1.17946]
         [5.50508, -5.97522, -2.35687]
         [-0.969843, 2.73667, 4.7403]
         [-2.38129, 4.79214, -6.1126]
         ⋮
```

Simulating an ideal gas

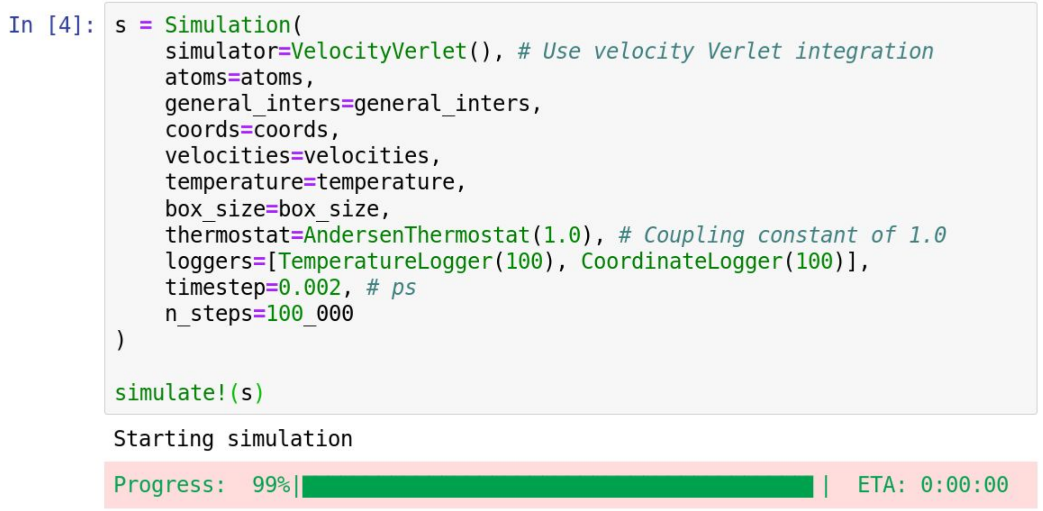
- Now we can define our dictionary of general interactions, i.e. those between most or all atoms
- Because we have defined the relevant parameters for the atoms, we can use the built-in Lennard Jones type



```
In [3]: general_inters =  Dict("LJ" => LennardJones())

Out[3]: Dict{String,LennardJones} with 1 entry:
          "LJ" => LennardJones(false)
```

Simulating an ideal gas

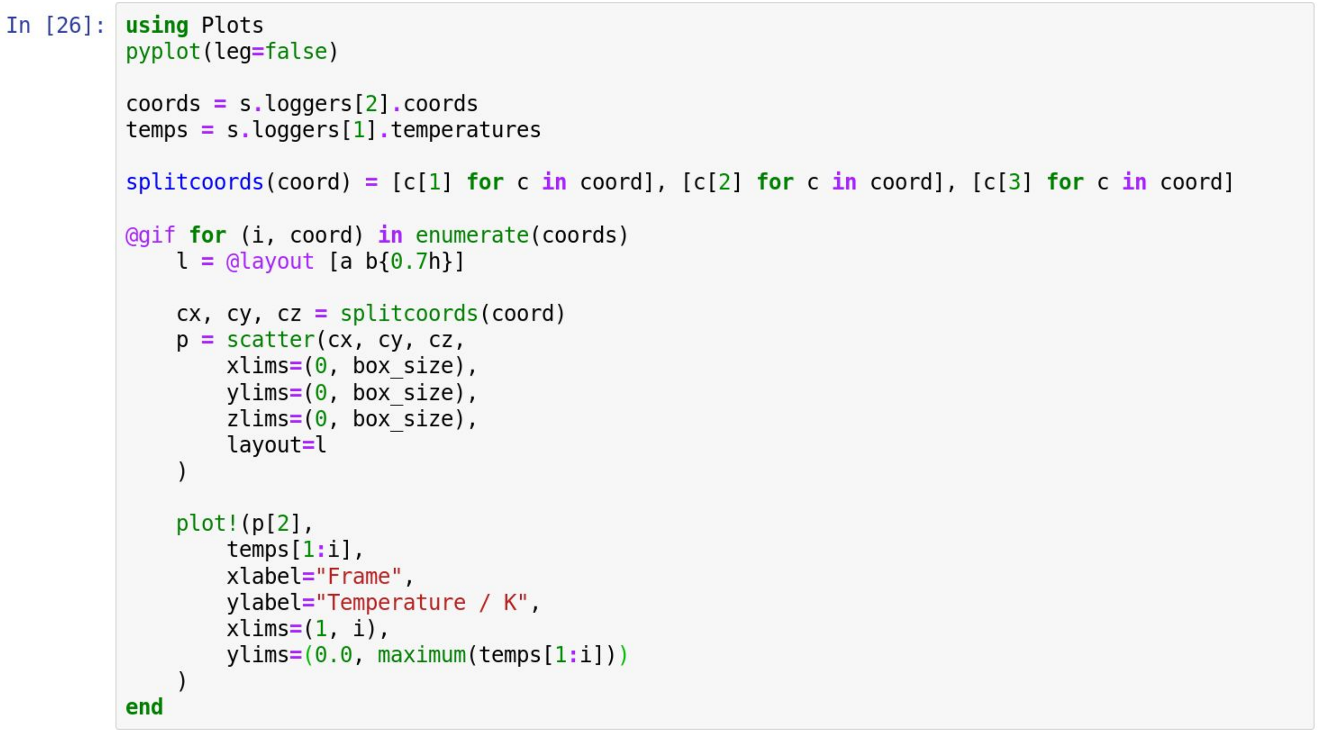
- Define and run the simulation
- Use an Andersen thermostat to keep a constant temperature
- Log the temperature and coordinates every 100 steps



```
In [4]: s = Simulation(
            simulator=VelocityVerlet(), # Use velocity Verlet integration
            atoms=atoms,
            general_inters=general_inters,
            coords=coords,
            velocities=velocities,
            temperature=temperature,
            box_size=box_size,
            thermostat=AndersenThermostat(1.0), # Coupling constant of 1.0
            loggers=[TemperatureLogger(100), CoordinateLogger(100)],
            timestep=0.002, # ps
            n_steps=100_000
        )

        simulate!(s)

        Starting simulation

        Progress:  99%|███████████████████████████████████████| ETA: 0:00:00
```

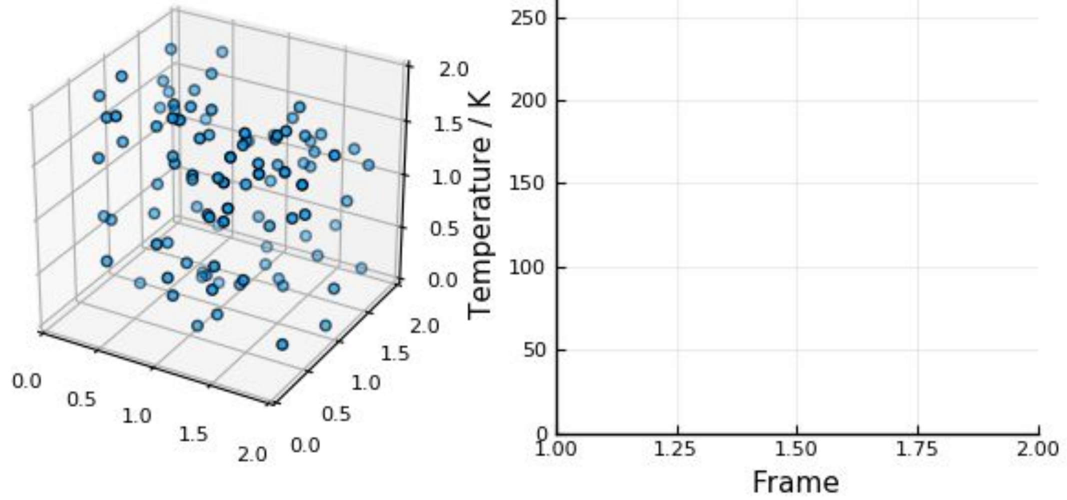
Simulating an ideal gas

- Plot the simulation



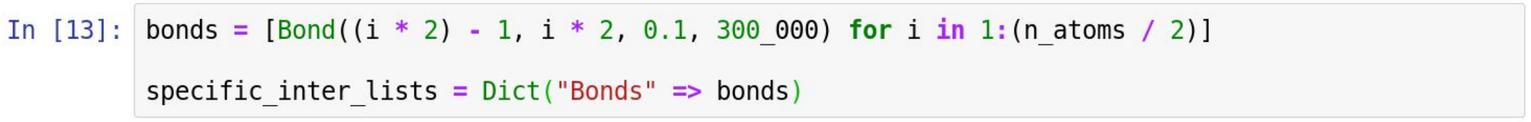
```
In [26]: using Plots
         pyplot(leg=false)

         coords = s.loggers[2].coords
         temps = s.loggers[1].temperatures

         splitcoords(coord) = [c[1] for c in coord], [c[2] for c in coord], [c[3] for c in coord]

         @gif for (i, coord) in enumerate(coords)
             l = @layout [a b{0.7h}]

             cx, cy, cz = splitcoords(coord)
             p = scatter(cx, cy, cz,
                 xlims=(0, box_size),
                 ylims=(0, box_size),
                 zlims=(0, box_size),
                 layout=l
             )

             plot!(p[2],
                 temps[1:i],
                 xlabel="Frame",
                 ylabel="Temperature / K",
                 xlims=(1, i),
                 ylims=(0.0, maximum(temps[1:i]))
             )
         end
```

Simulating an ideal gas

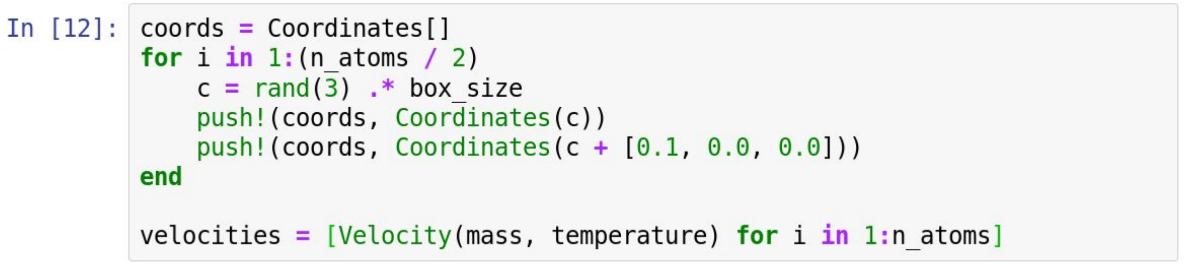


Simulating a diatomic molecule

- We can define specific interactions between groups of atoms, e.g. bonds
- Use the built-in bond type to place a harmonic constraint between paired atoms



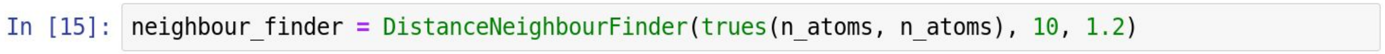
```
In [13]: bonds = [Bond((i * 2) - 1, i * 2, 0.1, 300_000) for i in 1:(n_atoms / 2)]

         specific_inter_lists = Dict("Bonds" => bonds)
```



```
In [12]: coords = Coordinates[]
         for i in 1:(n_atoms / 2)
             c = rand(3) .* box_size
             push!(coords, Coordinates(c))
             push!(coords, Coordinates(c + [0.1, 0.0, 0.0]))
         end

         velocities = [Velocity(mass, temperature) for i in 1:n_atoms]
```

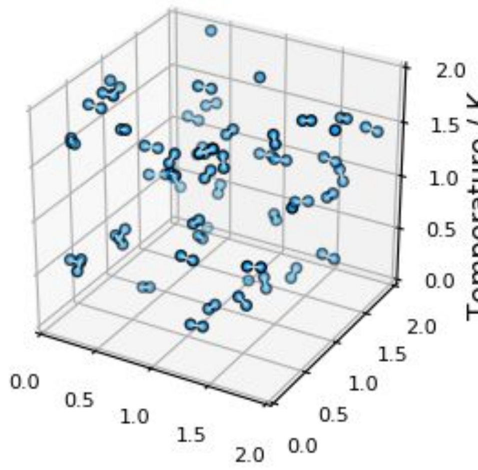
Simulating a diatomic molecule

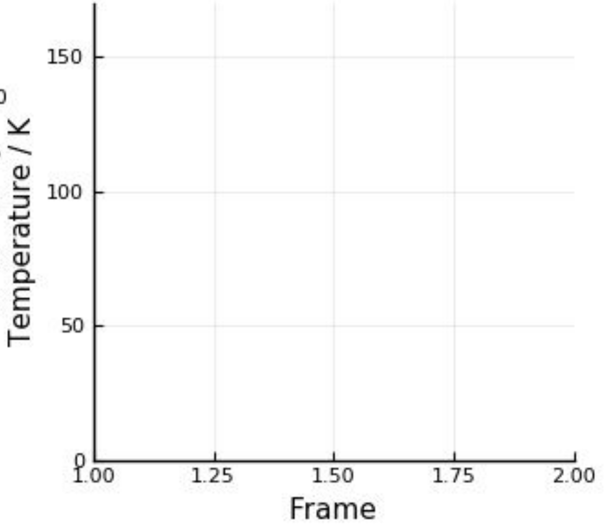
- Use a neighbour list to speed up the Lennard Jones calculation
- We will use the provided neighbour finder but you could write your own



```
In [15]: neighbour_finder = DistanceNeighbourFinder(trues(n_atoms, n_atoms), 10, 1.2)
```

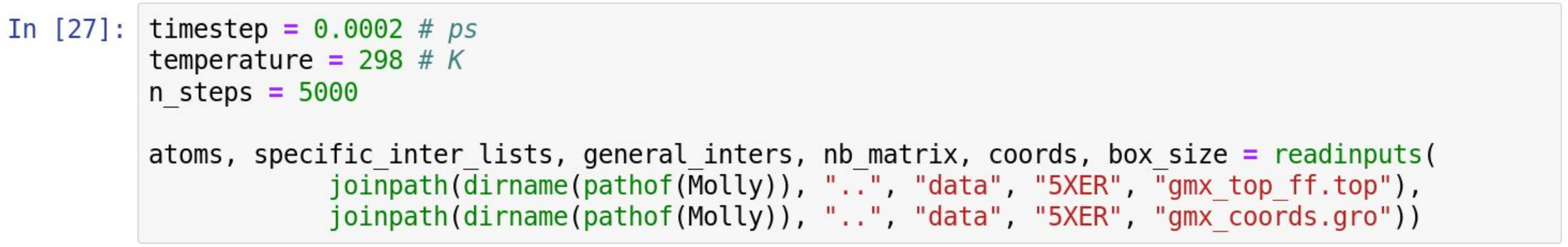
Simulating a diatomic molecule





Simulating a protein

- In addition to Lennard Jones forces and bonds, proteins also have angles, dihedral angles (torsions) and electrostatics
- Molly.jl has basic functionality to read in topology and forcefield data from GROMACS format files

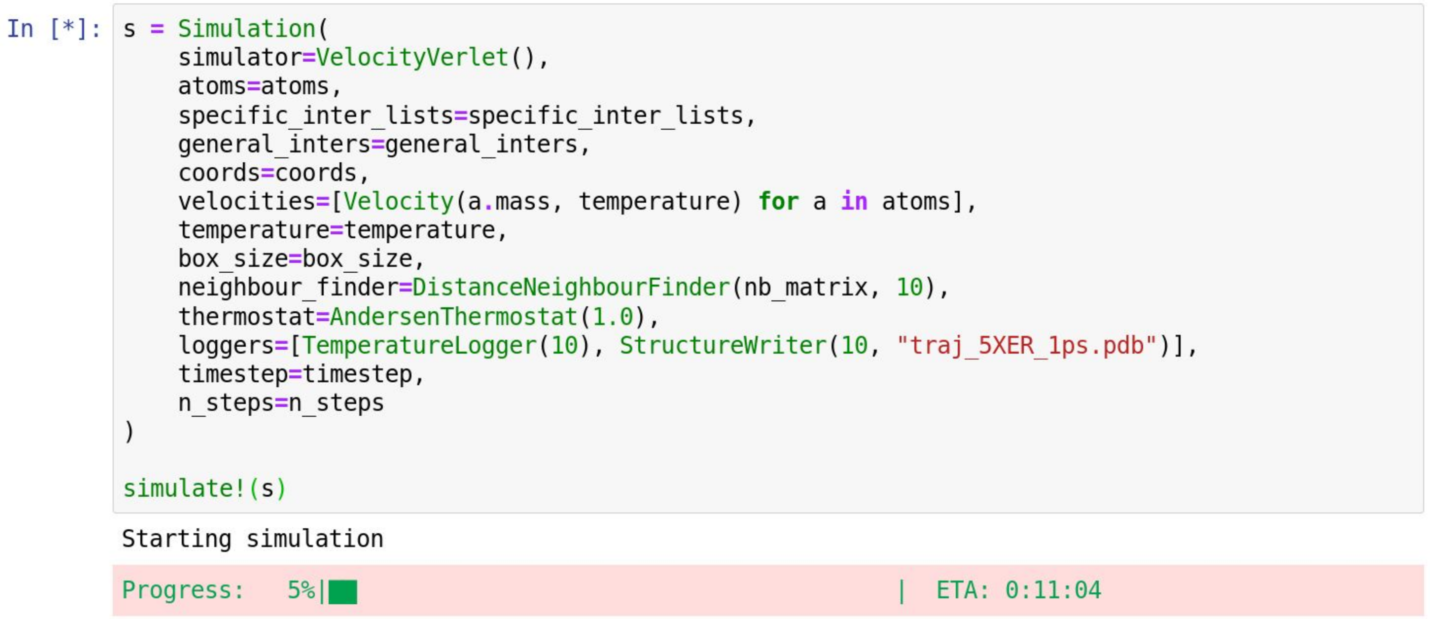


In [27]:

```
timestep = 0.0002 # ps
temperature = 298 # K
n_steps = 5000

atoms, specific_inter_lists, general_inters, nb_matrix, coords, box_size = readinputs(
            joinpath(dirname(pathof(Molly)), "..", "data", "5XER", "gmx_top_ff.top"),
            joinpath(dirname(pathof(Molly)), "..", "data", "5XER", "gmx_coords.gro"))
```

Simulating a protein

- Simulate as before

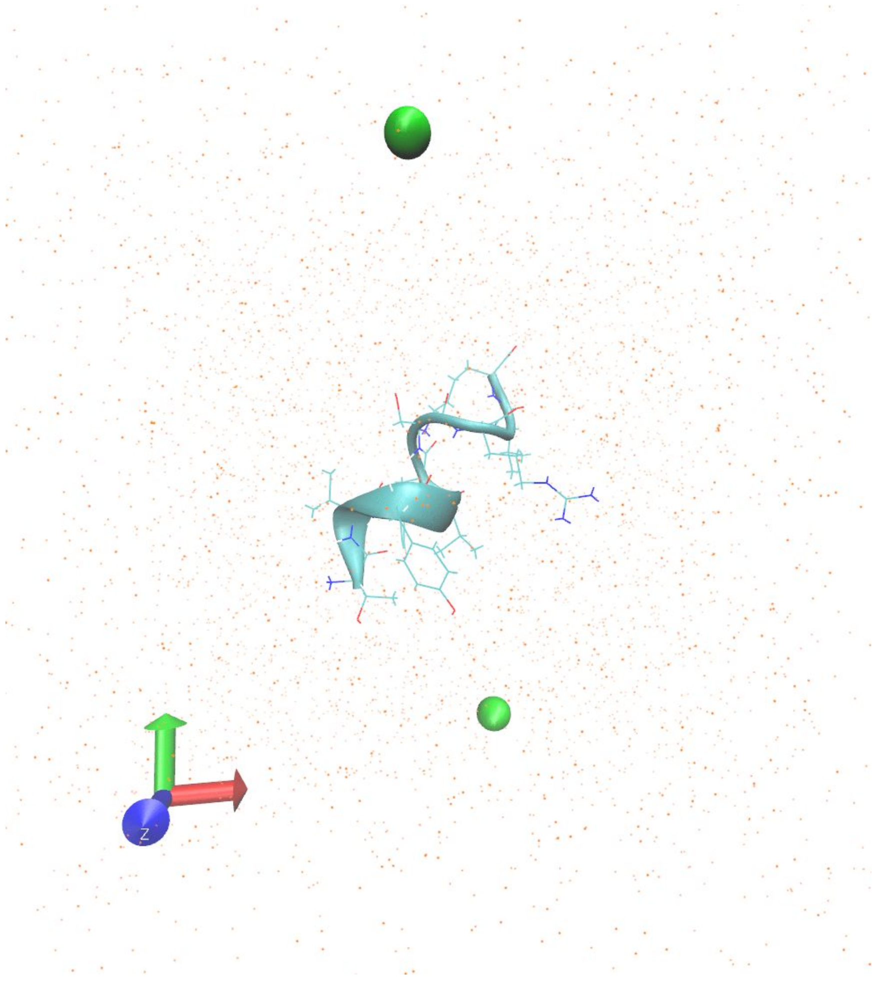


```
In [*]: s = Simulation(
            simulator=VelocityVerlet(),
            atoms=atoms,
            specific_inter_lists=specific_inter_lists,
            general_inters=general_inters,
            coords=coords,
            velocities=[Velocity(a.mass, temperature) for a in atoms],
            temperature=temperature,
            box_size=box_size,
            neighbour_finder=DistanceNeighbourFinder(nb_matrix, 10),
            thermostat=AndersenThermostat(1.0),
            loggers=[TemperatureLogger(10), StructureWriter(10, "traj_5XER_1ps.pdb")],
            timestep=timestep,
            n_steps=n_steps
        )

        simulate!(s)
```

```
Starting simulation
Progress:   5%|█                                           |  ETA: 0:11:04
```

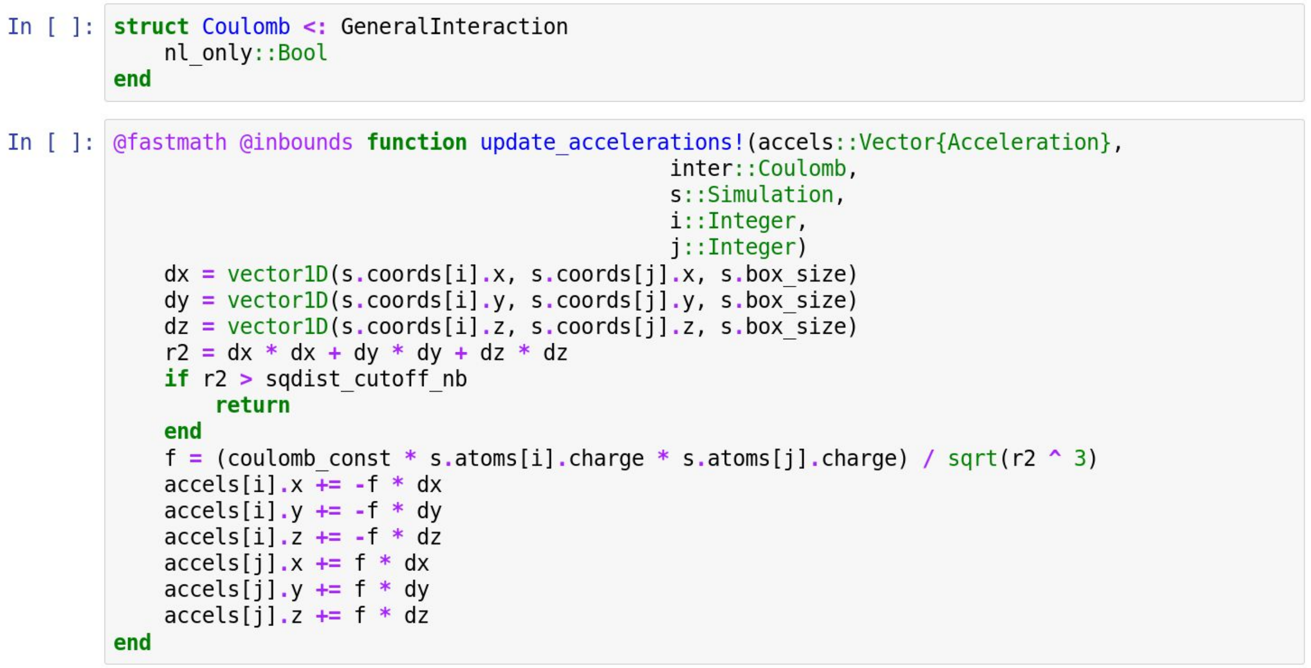
Simulating a protein

- 1 ps simulation viewed with VMD



Defining a force

- The entire definition for the electrostatic (Coulomb) force

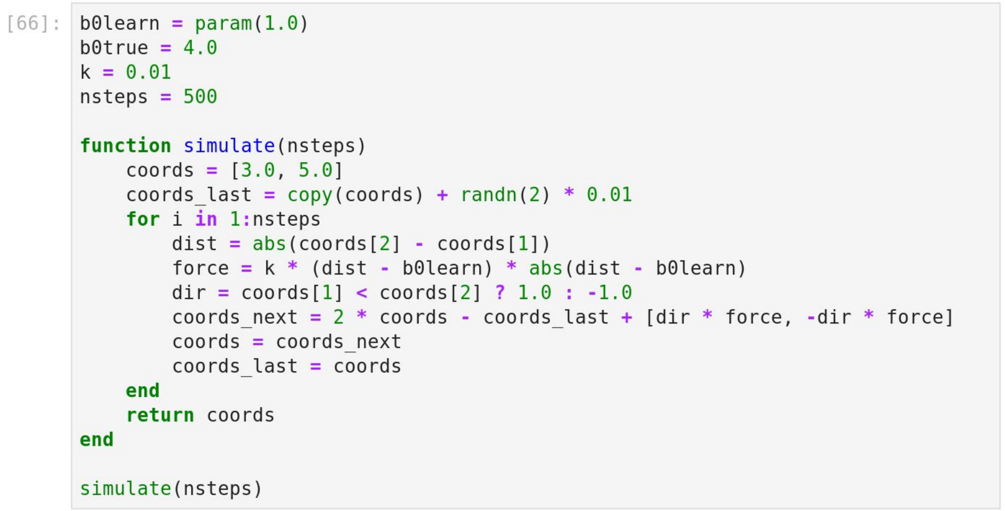


```
In [ ]: struct Coulomb <: GeneralInteraction
            nl_only::Bool
        end
```

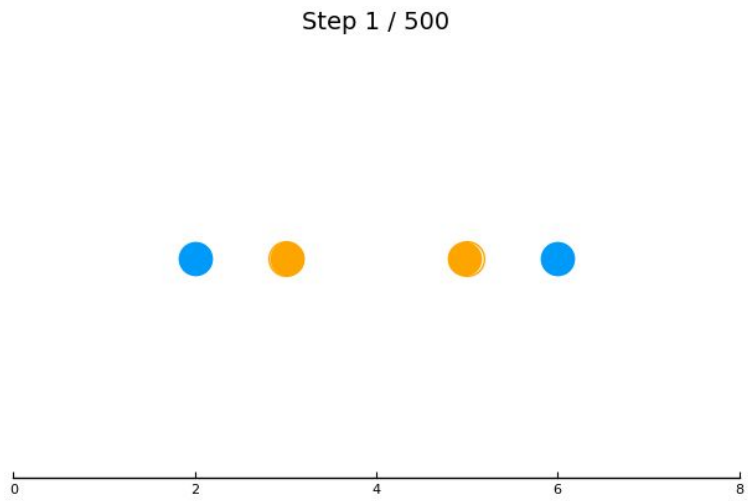
```
In [ ]: @fastmath @inbounds function update_accelerations!(accels::Vector{Acceleration},
                                                            inter::Coulomb,
                                                            s::Simulation,
                                                            i::Integer,
                                                            j::Integer)
            dx = vector1D(s.coords[i].x, s.coords[j].x, s.box_size)
            dy = vector1D(s.coords[i].y, s.coords[j].y, s.box_size)
            dz = vector1D(s.coords[i].z, s.coords[j].z, s.box_size)
            r2 = dx * dx + dy * dy + dz * dz
            if r2 > sqdist_cutoff_nb
                return
            end
            f = (coulomb_const * s.atoms[i].charge * s.atoms[j].charge) / sqrt(r2 ^ 3)
            accels[i].x += -f * dx
            accels[i].y += -f * dy
            accels[i].z += -f * dz
            accels[j].x += f * dx
            accels[j].y += f * dy
            accels[j].z += f * dz
        end
```

Differentiable molecular simulation

- Recent developments with Flux.jl make it possible to run molecular simulations and obtain gradients of parameters, e.g. forcefield parameters

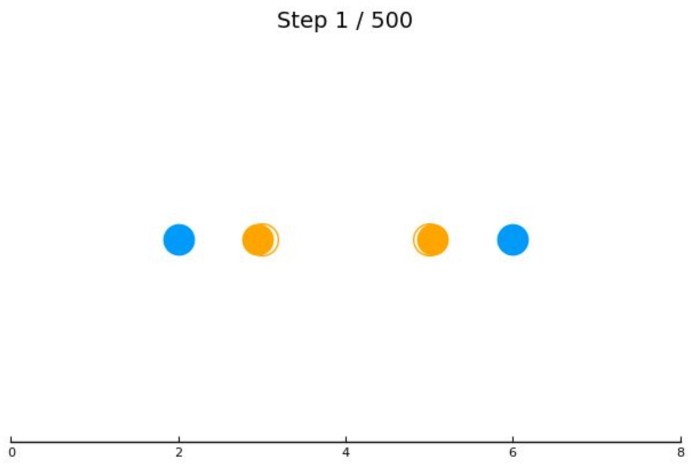


```
[66]: b0learn = param(1.0)
      b0true = 4.0
      k = 0.01
      nsteps = 500

      function simulate(nsteps)
          coords = [3.0, 5.0]
          coords_last = copy(coords) + randn(2) * 0.01
          for i in 1:nsteps
              dist = abs(coords[2] - coords[1])
              force = k * (dist - b0learn) * abs(dist - b0learn)
              dir = coords[1] < coords[2] ? 1.0 : -1.0
              coords_next = 2 * coords - coords_last + [dir * force, -dir * force]
              coords = coords_next
              coords_last = coords
          end
          return coords
      end

      simulate(nsteps)
```



Differentiable molecular simulation

```
[74]: function loss(b0true, nsteps)
          coords = simulate(nsteps)
          dist = abs(coords[2] - coords[1])
          return (dist - b0true) ^ 2
      end

      gs = Tracker.gradient(() -> loss(b0true, nsteps), params(b0learn))
      gs[b0learn]

[74]: -5.773390700589583 (tracked)

[75]: for i in 1:500
          gs = Tracker.gradient(() -> loss(b0true, nsteps), params(b0learn))
          update!(b0learn, -0.01 * gs[b0learn])
          i % 50 == 0 && println("Epoch ", i, " - b0learn ", b0learn)
      end

      Epoch 50 - b0learn 2.6801969855276746 (tracked)
      Epoch 100 - b0learn 3.574286748329535 (tracked)
      Epoch 150 - b0learn 3.9043047080232394 (tracked)
      Epoch 200 - b0learn 4.025176930121734 (tracked)
      Epoch 250 - b0learn 4.069391236713611 (tracked)
      Epoch 300 - b0learn 4.0855609902908805 (tracked)
      Epoch 350 - b0learn 4.09147724816171 (tracked)
      Epoch 400 - b0learn 4.093635443806043 (tracked)
      Epoch 450 - b0learn 4.094422175776826 (tracked)
      Epoch 500 - b0learn 4.094708733260627 (tracked)
```



Julia for computational biology

Pros

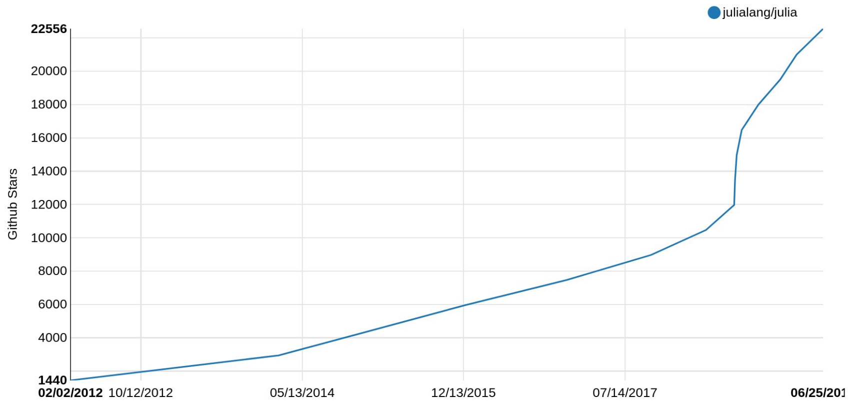
- High-level language with nice syntax
- Fast
- Prototype and code in the same language
- Sane design
- Great community
- Making packages is relatively easy
- There is a rabbit hole to go down

Cons

- Compile time requires a workflow change
- Performance gotchas
- Effort required to shift from OOP
- Plotting is still getting there
- Libraries still being implemented
- Sometimes you have to go down the rabbit hole

The future

- Popularity of the language increasing
- Increasingly used in teaching and research
- Stability of Julia v1.0 has made things easier for package developers
- Exciting developments in the language:
  - Improved multithreading
  - GPU/TPU programming
  - Deep learning/differentiable programming
  - Static compilation



Acknowledgements

- UCL Bioinformatics Group
- European Research Council
- Julia community
- BioJulia, in particular:
  - Ben Ward (GitHub BenJWard)
  - Kenta Sato (bicycle1885)
  - Joel S (joelselvaraj)