1.1 lab session notes

Potential plan (written 21/01/2026):

- Lab induction
- Dr Turci walks us through introduction
- Discuss refactoring code to give us a "complete virtual lab apparatus":
- A complete system which we can use to generate arbitrary configuration datasets for study
- Setup github repo
- Start refactoring/writing code (hopefully finish this quickly - that way rest of lab can be running simulations w/minimal need for more coding)

Reasoning for this:

- What is needed from code seems fairly clear at this point; need to be able to generate arbitrary configurations simulated to equilibration; need to be able to extract arbitrary observables; need to be able to assemble these into datasets over parameter variation
- Currently still going through "percolation" chapter [**?**] - understanding of theory isn't fully there yet, however understanding of code requirements ~ is - therefore start with what is known, and familiarise self with theory more later

Lab log (22/01/2026):

Turci introduced us to project: * Gave us overview

Met up w/lab partner, are trying to figure out project direction and logistical questions: * Setup shared google drive folder to share materials * Setup planning document * General starting aims: * Find critical exponents * Study cluster behavior * Vary over temperature and dimensionality * Agreed to refactor code in terms of config object * Object oriented config object w/pickle library save/load functionality - given that sims may be computationally intensive, saving results for future reanalysis may save compute time * Decided to use inheritance structure; parent config, and 1D, 2D, 3D, 4D config children - sim steps will look different in different dimensions and may be difficult to generalize - hence inheritance structure

Talked to Turci: * Need to figure out when state converges

- Setup code refactoring thus far:
  - Parent config class - has saving/load methods
  - Setup 2D config as test child case

Have setup code in object oriented approach: Abstract config class 2D config class Have done rudimentary testing

To do list: * FIX STATE LIST

- Clean up code
- Implement higher and lower dimensions classes
- Read up and figure out critical exponents
- Optimisation and factorisation:
  - Figure out numba python library; speed up system
  - Maybe rework observable lists into pandas dataframe? May be more generalisable/easier to work with
  - Double check observable calculations
  - Add "average observables" - need to be able to average over several iterations
  - Maybe save multiple config steps?
- Potentially setup "dataset" object? Set of config objects along some varied quantity? Save to folder of
- Setup github quarto lab book; switch over to lab book

##Lab log (24/01/2026):

- Decided to switch back to original state system for now; ik it works; may need to refactor it later
- Gonna try and implement numba on 2D case; test everything in 2D, then can expand out to higher dimensions
- Nvm; gonna try and get good bones in, and save all states

For current use, two general test cases:

- standardTest2000 = Ising model 100x100 cells run for 2000 steps
- standardTest500 = Ising model 100x100 cells run for 500 steps
- Have implemented saving all states - it does make the saved files significantly larger (I.E, 100x100 model w/100 steps, saving energy/magnetization observables is 32.5Mb) - quick test standardTest2000 s; took several minutes (didn't time it, but took a while), saved to 192.5 Mb
  - Currently that is acceptable; (given 100x100x2000 = 2x10^7 stored cells)) that suggests each cell uses ~20 bytes of storage space; so extrapolating to a 3D case 100^3 2000=40Gb - a 4D case may be prohibitive w/~4Tb storage space for the same size;

        * Could add discard functionality (I.E, don't need the initial equilibration steps)
        * Could only save the observables over that simulation length
        * However for now having the default "save all raw data" option is good to have

- Thinking of optimisations:
    - mcMove method
    - calcEnergy method
    - Both are VERY inefficient; in similar ways
    - Both run through entire configuration; both check adjacent cells; currently implemented through for loops; however some steps may be more efficiently done by either numpy array operations, or numba implementations
    - It may also be interesting to see if adjacency checks could be generalised/factorised to any dimension; if that's possible, inheritance structure becomes unecessary, as the algorithm could be implemented for all dimensionalities
        * It would be very useful if generalising is possible; this would allow detailed study of higher dimensional ising models
    - Config state storage - currently an array of integer values; am wondering if can switch to a binary representation - would make storage of saved files more efficient (I.E, 4D 100x100x2000 case could optimistically be reduced to 200Gb)
    - Data discard system - options to discard all but the N most recent configurations - storage data optimisation

If its possible to generalise the dinemsionality of the model; that opens up more lines of investigation

Unsure which optimisations are worth it. Plan of action:

- Try and get numba working
- Familiarise self w/numba operations
- Try and refactor/optimise above with numba in mind

Got numba working; doing rudimentary tests to see if it improves performance:

- Adding numba decorator to mcMove method
- standardTest500 yields Dt = 96.8 s
- standardTest500 w/out numba yields: 92.2 s
- This suggests no significant speedup
    - Suggests I'm using it incorrectly
- Testing standardTest500 w/ provided code;
    - w/out numba: 38.8 s
    - W/ numba: 4.26 s

- Maybe object oriented approach is breaking stuff?
  - Changed mc move; moved all self checks to runSimulation instead; that way mcMove is more “anonymous” and doesn’t need to call self
    - standardTest500: 47.8 s
    - Significant improvement
  - Wondering if difference between provided code and own code is due to “additional” code run by “simulationRun” - have commented out plotting and appending observables
    - standardTest500: 7.3 s
- Takeaways:
  - Since Numba compiles individual functions to machine code; it is advisable for numba accelerated functions/methods to be “streamlined” w/out needing to refer to non accelerated code
- Unsure if numba accelerated functions can refer to other accelerated functions
  - Test case: adjacency code in mcMove; move into separate accelerated method
    - Can move it to separate function, but numba gets confused if it’s a class method
    - standardTest500: 6.9 s

Numba seems to work; moving onto some architecture changes: * Current plan is to have inheritance structure w/specific dimension configs hard coded * 3 main differences between dimensions: Dimension of array First initial state Adjacencies of cells * Cell adjacency seems the most complicated to generalise; however tuple indices may allow for * Numba is DIFFICULT to work with; it has the type stringency of a C language w/out significantly useful error messages

FINALLY got the numba stuff working; still needs optimisatin; however everything is working in a generalised sense; I think a general N dimensional config object is doable now:

- standardTest: 130.6 s
- This is significantly worse than before; I thing its because of the generalised adjacency changes? Alternatively my laptop is on low battery with other stuff running on background; so may be that instead?
- TBD another day
  - If it’s a genuine slowdown, may need to comb through and see if its possible to optimise
  - Adjacency code - may need to do that once per dimension case? May help reduce computation? Idk

25/01/2026

Quick test; still slow on full battery/restarted laptop; needs optimisation

- spliting adjacency function "getNeighbourIndices" into two:
    - Get adjacency maps (array of addition/subtractions to be made) - can be done on basis of dimension once per config object
    - Adjacency maps can then be passed into "getNeighbourIndices" to add maps%size to desired index
    - Splitting computation should reduce repeat workload of finding adjacency maps
- getArrayVal; can rewrite for loop in terms of array manipulations; unsure if it'd be faster given compilation, but numpy may have more efficient implementations
    - Maybe setup some default D=1,2,3,4,5 search cases; may be faster for defaults that are likely going to be used in this lab; then remaining code
    - getArrayVal uses a size^{currentDimension} array; it's the same for configs of same dimension/size; precomputing this for each config may help (instead of recomputing it for each
    - flattenArray step may be movable to "getNeighbours" function instead? Demodularises the code, but would reduce repeat operations - could also pass in dimensionality as argument instead of evaluating it locally (again reducing repeat steps)
- Maybe combine several functions - I wonder if calls between numba'd functions reverts to python, requiring recompilation and slowing down results

Plan:

- Combine config2D into config; code should work generalised:
    - Careful w/2D indices in mcMove and calcEnergy; need to generalize this
- Ensure config object has information about its dimensionality
- Have config object query its own adjacency maps in the constructor (to avoid repetition)
- Have adjacency maps passed into Mcmove, and into getNeighbourIndices
- Combine functionality of getArrayVal into getNeighbours - leave getArrayVal as is for now (may be useful for some general one off computations) but streamline getNeighbours for efficiency as outlined above
- Test
- If need be, maybe combine getNeighbourIndices into getNeighbours - if call between functions is slowing numba down, this is an efficiency gain (at the cost of modularity and legibility)

  - If need be combine this all into mcMove directly; preffer not to; modular code tends to be more legible, but it may be a worthwhile tradeoff; esp given that this code is the "core" of the simulation, and shouldn't need to be touched once its working properlyi

Log:

- Problem - editing state Nd array is problematic w/numba
  - Workaround - don't edit Nd array in mcMove; have mcMove return list of positions and associated values to run simulation; run simulation can then handle the editing
    * Have found workaround; involves array.reshape(tuple) instead of np.reshape(array, array) - does require passing through sizes array as tuple; but that can be saved once by the config
- Have moved some functionality around
- standardTest500; 83s - better but still not good
- Edited main loop in Mcmove to loop over cells in config rather than over I, j - generalises loop
- standardTest500; 8.8s - seems that editing main loop vastly improved performance
- Quick test of 1D; broke - have noted #TODO in code

26/01/2026 log:

- Seems that editing main loop didn't iterate over all elements?
  - Fixed it; idk if there was actually a speedup
  - standardTest500; 96s
- Have yet to separate adjacency check out
  - Have seperated out "getNeighbourIndices" into "getNeighbourIndices" and "getAdjacencies" - "getAdjacencies" finds the relative neighbour indices (I.E, [0, +-1]) for a given dimensionality - this is done once per config object and passed through to "getNeighbourIndices" - should reduce computation
    * Slight improvement: standardTest500: 92s
- Wondering if "nested function calls" is whats causing issues?

  - Quickly moving everything into mcMove - is a lot less modular, but may be more efficient, so worth testing standardTest500: 87.4s Slight improvement - I suspect not duplicating the array flattening step is improving things More tweaking: standardTest500: 81.2s
    * More tweaking:
    * standardTest500: 62.1s
    * More tweaking (merged editArray into mcMoves)
    * standardTest500: 33.3s
    * More tweaking - changes how e^(−cost)= (e $^{(-))}$(cost) is computed - precomputing (e ^(−)) to try and shave some more efficiency
    * standardTest500: 50s; have reverted change - presumably multiplication and one exp is easier on computer than alternative
    * Minor testing; commented out "appendConfig(curConfig)" from "runSimulation" - given 500 steps, that gets called a lot; want to quantify how significant a drain it represents
    * standardTest500: 23.4s Does represent a not insigificant improvement May be including a "do not save" option to runSimulation - have added - should help w/both run times and file sizes

- Taken a break from optimisation to debug; have gotten it working in 1D case
- General debugging complete; have quickly tested that 1D, 3D and 4D cases work; they do; have added graphing method to handle 3D case and rearranged graphing code a bit
- Code still needs commenting/general tidying, but it is in a baseline "complete" state
- Note; observing significant decrease in speed for higher dimension; setting up new test case:
- standardTest3D100 = 20x20x20 over 100 steps
- standardTest3D100: 562.6s (9.3min)
- This is a significant increase from a 2D case; I suspect this will be significantly worse for even higher dimensions; as such it may be prudent to try and understand finite scaling effects in lower dimensions, and try and extrapolate that understanding to small higher dimensionality models as necessary
- Note; incorrect implenentation of code (removing np.flip(currentIndex) operation on line 215) creates an incredibly interesting behaviour which includes:
  - Symetry along the x/y axis 3 distinct cluster types; black, white, crosshatched