10/21/18 Experimenting with my 1D python DD code:

Testing speedups due to different modifications:

Running code with same parameters as the matlab and c++ codes and -0.5 to 1.2V and only outputting the JV data and no excessive output to terminal.

Original time: 143sec

1. Adding @jit (from numba) to Thomas\_solve

Sped up to 103sec

**Note: the fast version of numba, with the nopython=true tag, raises exception meaning my function can’t be optimized that way! Same goes for all other functions tried to add @jit with nopython tag, it gave an error.**

2. Try adding @jit to other functions, i.e. continuity setup, poisson setup, Bernoulli… With all of these, the cpu speed actually slowed, to ~120sec.. So I guess the time to compile is taking longer than before..

So removed @jit from these functions.

3. checked if in Thomas\_solve, when using @jit, directly accessing the diagonal arrays through system. (the object) makes a time difference:

**Yes, large slowdown!, to 152 sec from 103 sec!**

So it is better to make a copy (i.e. I was using deepcopy) of the arrays and then use those…

**Note: if access the arrays directly using the objects with numba, it is actually slower than without adding the @jit tag! (w/o @jit was 138 sec).**

**4.** Check if the np.copy() is faster than using copy.deepcopy for within Thomas solver:

This is for copying the diagonal, upper, lower, and rhs arrays.

Same time: 103sec 🡪 the copying seems not the bottleneck…

5. Check if passing explicitely these arrays to the Thomas\_solve function, instead of passing an object is faster.

**But I still need to do the copy for the diagonal, otherwise it will be modified by the thonmas solve! Rhs is ok to be changed, b/c it is resetup in every iter anyway…**

**Yes, that is faster!!: 96.6 sec🡪 makes sense that passing less stuff (i.e. not the entire object) is faster!**

**And now it is able to compile with nopython=True!! 🡪 it was the objects passing that gave issues! But found that adding nopython=True didn’t give additional speedup, so probably it was doing that already anyway!**

Now let’s profile it, to see actually how much speed up we gained within Thomas solve…

**Previous, first implementation had 43sec spent in Thomas\_solve.**

**Now 537 ms!!!!!!!!!!!!!!!!!!!!!!!!!!!!**

THAT’S AN 80x speedup!!!

Now need to work on the other functions which are the bottleneck..

5. **Sped up bernoulli functions by a lot by passing the bn1 and bn2 arrays themselves and using them within the for loop, instead of using self. to access the member variables within the loop. Also added @jit.**

**note: since python passes variables like by reference, if the variables are mutatable, like an array, we can mutate them inside a function and the changes will remain outside the function! So we don’t need to update self.B\_n1, b/c just by changing the passed in self.B\_n1, we are mutating the actual object attribute.**

**Confirmed that the speedup was due to not using self. within the for loop, not just due to the @jit!!**

**Went from spending 25 sec in the Bernoulli\_n function to 129 ms!!!!!**

**A 193x speedup!!!**

**Try to move Bernoulli function outside of the continuity class to see if that will let me use the @jit(nopython = True) option or if will give any additional speedup.**

**Yes, it does allow me to run nopython = True!!!-> so I figured that part out!!!--> need to have no objects in the way, if want to compile the functions!!! With numba and nopython = true!!**

**Again, comparing explicitely is there a time difference between having Bernoulli function inside the class and outside??:**

**Yes, it’s a bit faster outside the class: 52.6 sec when inside class, 50.4 sec when outside. These are total times..**

**We are at 50.1sec total right now. Already almost a 3x speedup!!**

**Now want to try to adjust the other functions to see if can improve further!**

**The next highest bottleneck is: set\_main\_diag of the continuity equations modules at 5.5 sec each**

**So use the same strategy, pass to the set\_main\_diag function all the necessary variables, instead of reaching in and using self….., so then can use @jit for speedup.**

**Yes, that worked: went from 5.5sec to 325ms!**

**Now we are at 43.89sec**

**Now follow the same strategy for setting of upper and lower\_diags:**

**Now we are at 35.2 sec**

**Did similar stragety with compute R Langevin, but was trickier b/c it was relying on stuff from params…**

**So I had it grab the stuff from params once, instead of doing it within the for loop. I find grabbing stuff form objects within for loops is very slow!!**

**Now are at 29.9 sec which is equivalent to the Matlab code speed!**

But now most of the cpu time is in typeof\_pyval which is related to the numba compiling.: not sure how to speed that up!

**Set rhs of continuity eqn’s is a bit slow at 2sec, try to speed that up…**

**Trying to speed it up, actually slowed things down!!!--> the set rhs went down to 400 ms, but gained several seconds on the typeof\_pyval… calls..**

**So go back to previous**

**Tried passing directly to R\_Langevin the rcombo parameters from main🡪 makes it a big faster.**

**Now at 29.2 sec**

**So I have achieved a 5x speedup. So properly selecting how to write the code is important!**

**10/21/18 5pm continued:**

1. Try adding parallel = True to jit spots:

Seems that actually adds overhead and makes it slower!

**Actually much slower🡪 down from 31 sec to 42 sec!!**

**2. Let’s try comparing with a smaller dx of 0.25nm, to see if the larger simulation will comopensate for the numba overhead.**

**b/c right now most of the cpu time is spend on the type\_pvval which is the numba overhead!**

**So change in matlab and c++ and python to 0.25nm grid, and rerun for this new comparison:**

**And just use gen rate = constant = the gen rate max… so don’t need to provide new gen rate…**

**With the larger system, python loses to Matlab!:**

**Timings are:**

**Matlab: 40.7sec**

**Python: 101 sec**

**C++: 3.4 sec**

**Notice how both the python and c++ code slow down was about 3.3-3.7x, but matlab only slowed down by 1.3x 🡪 it is somehow doing things very fast!**

**3. Check if the jit at the set main, upper, lower diag is actually helping speed things up or too much overhead:**

**Comment them out and profile.**

**With jit, we spend 14.71 sec in setup eqn function.**

**Without jit, we spend 52.9 sec🡪 so the jit significantly speeds things up!**

**NOTE: MATLAB USES JIT!!!--> SO IT IS A FAIR COMPARISON TO USE JIT IN PYTHON ALSO!!**

**4.**

**Try this for the set main diag definition:**

**i.e. do the multiplications as precalculate array operations, instead of within the loop!**

**@jit**

**def set\_main\_diag(self, main\_diag, p\_mob, B\_p1, B\_p2):**

**tmp1 = p\_mob\*B\_p2**

**tmp2 = p\_mob\*B\_p1**

**for i in range(1, len(main\_diag)):**

**main\_diag[i] = -(tmp1[i] + tmp2[i+1]) #-(p\_mob[i]\*B\_p2[i] + p\_mob[i+1]\*B\_p1[i+1])**

Try this strategy for main, upper and lower diag in continuity\_p, while leaving continuity\_n the same as before.

See if there is any speedup.

**Seems there’s not any noticeable speedup here…🡪 this is probably not the bottleneck!**

**Try moving the set functions out of the class for continuity\_p and add nopython = True to them:**

**Got a little speedup: down to 97sec from 99sec on total time.**

**Add parallel = True also to the set functions also:**

**Parallel made it way slower!!--> down to 132 seconds!**

**Move the set functions back inside the class, remove the nopython = true.., but leave the jit.**

**See if significant slowdown…🡪 if not then leave them in the class since is more elegant. It’s about the same: 100 sec in the class, 97 seconds outside.**

**Try to speed up set\_rhs which is quite slow now:**

**Yes that worked!!--> got down to 91 sec after doing this:**

@jit

def set\_rhs(self, Up):

self.rhs[1:len(self.rhs)] = -self.Cp \* Up[1:len(self.rhs)]

#for i in range(1, len(self.rhs)):

#self.rhs[i] = -self.Cp \* Up[i]

self.rhs[1] -= self.p\_mob[0]\*self.B\_p1[1]\*self.p\_leftBC

self.rhs[len(self.rhs)-1] -= self.p\_mob[len(self.rhs)]\*self.B\_p2[len(self.rhs)]\*self.p\_rightBC

**Now did the same with set\_rhs on the continuity\_n eqn:**

**Now is 87.2 sec for the 0.25nm version**

**The 1nm version now runs in: 31.2 sec🡪 so can’t see much improvement there🡪 but when deal with larger arrays, then see the improvement**