**Completed**

\*I find this valuable name confusing:

last\_snapshot = first\_snapshot + Nsnapshots

since this temporary variable is not necessary, just get rid of it

Done.

\* This comment confused me:

# Make a local copy of the header array augmented with list indices. We know

    # that the source array does not have any irrelevant entries, as these have

    # already been filtered out.

Initially I thought you meant there are no negligible processes. Now I am pretty sure I understand that this means that every entry corresponds to an actual reaction. If that is what you mean then please change the wording.

Changed irrelevant to ‘not corresponding to actual processes’.

\* No variable named oEF\_list\_old.

\* Only a single variable named oEF\_list.  It is clear that if this is before a snapshot in the code then it is from the previous snapshot. No different from kmc\_time in that respect.

\* Name variable tg.uEF\_list rather than  tg.uEF\_back\_calculated to be consistent with other namings.

I did see that you need to have ATF\_old versus ATF\_new when calculating the incremental throttling factors. I suppose it is probably okay to keep the old and new naming in those areas, and probably the rest of the code for now. So oEF\_list\_old and oEF\_list\_new is okay for now.

Changed oEF\_list to oEF\_TOF\_list\_next and oEF\_list\_old to oEF\_TOF\_list. The ‘\_next’ means for the next snapshot in the sequence. The ‘TOF” means it came from the unstructured TOF\_data\_list variable. **If you don’t like this extension, I’ll change it to whatever you prefer.** There was already a uEF\_list variable which has the uEFs for both process pairs together. I have renamed uEF\_back\_calculated to uEF\_TOF\_list to match oEF\_TOF\_list.

The below line does not appear to be extracting a header array:

    # Extract the header array

    unthrottling\_factors\_dict = tg.aggregate\_throttling\_factors\_dict\_old

Corrected to get the old throttling factors.

* What is the usefulness of having “set\_throttling\_case” and keeping track of “throttling\_case” ? I think this is unnecessary and just makes the code longer. For example, If there are no SPs, they will be stuck in throttling scale unthrottled with SP\_list having a length of zero. So I don’t see how there is any advantage of keeping track of this. Unless you can give me a compelling reason, it should be removed.

Done.

* Make a test case (no need to do it manually) for having only NSPs. See what happens.

Done. The ATFs are all set to 1. This means that the pre-exponentials aren’t modified and the next snapshot will be completely unthrottled. (I had to make a small fix to the testing runfile to initialize the new ATF dict, but this is unrelated to the rest of the code which would properly initialize it.)

* This variable does not seem to be used after being assigned: min\_dir
  + it should be removed

Done.

* This variable : min\_SRP\_EF = float(tg.Nsites)/tg.max\_time
  + should be made into a global, tg.NSP\_EF\_threshold.
  + Let’s recalculate it at same time other guidelines are being calculated.

Done.

Is get\_staggering\_Factors error free?

Note the following:

* whether the compression scale is 1 or 0, in both cases we receive fast\_idx = 0. To me this looks like some kind of problem.

Since the staggering factors are currently being pulled out of a function anyway, I don’t see any reason to have this function rather than just making the global variable look the way I originally wanted it:

tg.staggering\_factors = [ undefined, tg.Nsites, 10, 1.1]

that way scale 1,2, and 3 correspond to the indices. Unless you can find some compelling reason, just make it this way and get rid of the function. That function in the lines for calling it are just creating unnecessary temporary variables and code. Then we can just do this:

fast\_staggering\_factor = tg.staggering\_factors(compression\_scale[0])

slow\_staggering\_factor = tg.staggering\_factors(compression\_scale[1])

Deleted get\_staggering\_factors and converted most compression\_scale occurrences to either tg.fast\_throttling\_scale or tg.slow\_throttling\_scale. Changed tg.staggering\_factors to have an Nsites value in the first element (instead of undefined) and fast and slow staggering factors are now directly accessed in the staggering\_factors list.

* EF\_max was initially confusing to me. Please add a comment there that it is the maximum EF of that speedrank.

Done.

* In the loop for finding the BP: for i in range(BP\_index, SRP\_index + 1):
  + I found these variables confusing: EF\_ratio let’s make the variable names EF\_ratio\_i so it is clear that they are tied to the loop counter.

Done.

* around line 442 there seem to be something like 12 unnecessary lines, they appear after   
  “if fwd\_EF is not None and rev\_EF is not None:”
  + It looks like those can simply be reduced to:
    - EF\_max = MAX(fwd\_EF, rev\_EF).
    - But can’t you also just take EF\_max = tg.uEF\_list[master\_EF\_list\_idx][4] like you did in the earlier loop?

I believe you are correct. I believe that finding the SRP based on the process with the lowest uEF is also not correct; it should have been the process with the smallest max uEF. This code has been eliminated and EF\_max is now set from the master uEF list.

* compression\_scheme 🡨 change to compression\_schemes
  + also change the function set\_compression\_scheme to set\_compression\_schemes

Done.

find\_throttled\_EFs 🡨 why don’t we call this “calculate\_ptEFs” ?

Done.

successful\_scale\_compression 🡨 please change “outcome” to “compression\_achieved”, and change the function name to “check\_scale\_compression” since that is what the function does.

Done.

I guess that “uEF\_staggering\_factor = old\_uEF / current\_uEF” this is supposed to mean “last\_rank\_uEF / curren\_rank\_uEF” 🡨 this took me a while to figure out. It needs to be changed.

Done. I replaced all instances of old\_ in this context with last\_rank\_ and all instances of current\_ with current\_rank\_.

* Significant change: get rid of keeping track of the first throttled fast process (or first throttled slow process). We will keep track of the sFFP as well as rank 1 of the SPs, but rank 1 might be unthrottled. With our current algorithm, there is no need for implicit indexing, which is harder to ensure being written correctly.

This has been refactored to explicitly skip any ranks whose uEFs are within the FRP threshold (FFPs and SPs) or below the floor (FQPs).

Make the staggering factors have these names:

SF\_Max or SF\_compression for the staggering factor associated with the current compression level

SF\_Natural or SF\_Nat for the natural staggering factor (which is based on uEFs)

Done.

                                Beneath the following 2 lines, need an if statement similar to one shown below.

                                                                # Algorithm step 2.a

                fFQP\_uEF = ascend\_FFP\_list[sFFP\_index-1][1]

                                                                                if compression = no compression

                                                                                sFFP\_SF\_fFQP = min(tg.Nsites, sFFP\_uEF/fFQP\_uEF)

                                                                                else:

                                                                                sFFP\_SF\_fFQP = min(fast\_staggering\_factor, sFFP\_uEF/fFQP\_uEF)

Done.

* I have decided that just to make sure that there is no issue, whenever you are doing the “natural staggering check” versus “compressed check”, I want you to also do the “within Nsites of FRP” check at the same place. This is comparable to what we do with stepdown: running all the checks explicitly to make sure nothing has exceeded a buffer. This is easier to understand than some kind of implicit check to see if we are Nsites away from the FRP. For each FFP and SP compression attempt, just run that check at the same time you are comparing natural staggering to compression requirement. (i.e., if natural staggering is not enough, take FRP-buffer or compression, whichever is further from FRP).

Done. I also added a warning if compression to within Nsites \* FRP uEF happens more than once as it shouldn’t be doing that. After the first time, any further compression towards the FRP should stay outside the window due to minimum spacing buffers in place between processes.

\* Why does do\_throttled\_snasphot have so many things like "sps\_actual=..." ? I do not see the purpose of these.  This looks like unnecessary clutter.  If there is some compelling reason to update these before every snapshot, then it needs to be part of the update throttling globals function that gets called \*at the top\* of do\_throttled snapshot.

* Inside the snapshots module, make the achieved sps and tps into global variables (if they are not already) and then update them after a snapshot is run.
  + In the throttling module, printout\_throttling\_info should not have tps and sps as arguments. The achieved sps and tps should be called from the snapshots globals (just like the other things that are being printed there)
* What is the point of having variable assignments like these?
  + tg.BP\_index\_dict[ranking\_scheme] = BP\_index
  + if you think these should be global variables, just assign them within the function create\_FFP\_SP\_lists rather than returning them. That will make the code shorter.

Made lots of local variables into globals. Eliminated most/all shadowing of globals by locals by manipulating globals directly. Actual SPS and TPS are now calculated in do\_snapshot and stored in the snapshots globals. The actual SPS and TPS are still passed to printout\_throttling\_info but the names of the arguments have been changed to sps\_actual and tps\_actual. I did this to reduce somewhat the number of places/functions where the sg module is used in the throttling module.

\* Currently you are updating the snapshot variables inside the for loop of do\_throttled\_snapshots. This does not make sense to me. It should be inside of the do\_throttled\_snapshot function since this is part of every throttled snapshot. Otherwise need to provide a compelling reason why it is not inside that other function. <-- I am pretty sure I told you to put it there on Friday or Thursday. You need to keep track of every single specification given to you (even if it is one week apart) and make sure that what you produce matches those specifications.

\* Same thing for update\_throttling\_guidelines.

\* There is no reason for the update\_snapshot\_variables() to return anything. It should simply update the global variables that it needs to update.

\* There is no reason for this to occur inside do\_throttled snapshots rather than update\_snapshot\_variables:

sps = tg.throttling\_sps.

\* There is no reason to have this line after the snapshot is run, unless you are storing these for throttling\_info:  tg.proc\_names, tg.oEF\_list = update\_snapshot\_variables()

\* I think that saving the modules should also be inside inside do\_throttled\_snapshot. Singular. I don't see why it would not be.

\*\*\*

I put all these extra routines in do\_throttled\_snapshot and consolidated most updates into the two existing routines (update\_snapshot\_variables and update\_throttling\_guidelines). Module saving now occurs in do\_throttled\_snapshot.

You assigned sps\_local then you didn’t use it, you still used sps in the argument

Fixed.

Remove code after ‘# Initialize pre-exponential dictionary if needed’ from runfile.

Moved to apply\_throttling\_factors.

Remove code after ‘# Update the cutoff time’ from runfile.

Wouldn’t this be better if it was a function inside eil or something like that?

if load\_simulation\_state:

    # Load modules

    sg\_module.load\_params()

    tg\_module.load\_params()

    # Reset number of steps and time

    sg.model.base.set\_kmc\_time(sg.kmc\_time)

    sg.model.base.set\_kmc\_step(sg.steps\_so\_far)

    sg.atoms.kmc\_step = sg.steps\_so\_far

    # Load the lattice

    sg.model.\_set\_configuration(np.array(sg.config))

    sg.model.\_adjust\_database()

    # Read the PRNG state, if available, and set it in the model

    snapshots.seed\_PRNG(restart=True, state=sg.PRNG\_state)

    # Update the snapshot number

    tg.current\_snapshot += 1

It does not seem very user friendly.  I don’t think it’s good to encourage users to copy and paste something like this in runfiles rather than have them simply call a function when they want to load a simulation.

Created a new run initialization file to load old parameters and reset any values wrongly overwritten.

# Execute snapshots  I would like to have just a single line beneath this, not a loop, for example runfile.

**Need Feedback/Review**

* For the paired ranking (based on earlier guidance) you are not adding reverse processes that have a rate of 0. In the current algorithm, even those should be added to the ranked\_uEF\_list. A process with an event frequency of zero can still be throttled. [we don’t know if the event frequency will be nonzero in the next snapshot, so we still need to change the rate constant for it in our updated algorithm]. This is around line 250 in the code I am looking at, and is beneath # Add to temporary master list
* if ranking\_scheme == 'paired':

This is actually OK. The unranked uEF\_list has code (see get\_process\_info, about line 155) to reset the negligible process EFs to both be None instead of the uEFs (each process pair starts with its uEFs) based on a flag (throttle\_process\_pairs), and they are both reset to None only if both are less than the NSP threshold. And we still need the code around line 250 in case we have an explicitly irreversible process.

What does this mean? I am not sure I understand what this code is for, but ranks have to be sorted by the Max\_EF in that rank.  
 # Reverse the list of FFPs to be in proper ascending order. This sorting  
 # will break ties by putting forward reactions first.  
 ascend\_FFP\_list.sort(key=lambda x: (x[1], x[2]))

This code is OK. Each of the sublists in ascend\_FFP\_list is (0) the reaction number, (1) the rate associated with the process group (in this case a pair of processes), (2) the process name, and (3) the direction of maximum rate. The rate associated with the pair of processes is the max of the forward and reverse processes. I have updated the comment to better explain this.

* SRP\_EF = ranked\_uEF\_list[BP\_index][0][1] 🡨 I don’t think that we use this anymore. Was this used for creating EF\_range in the past? I think that we would use the slowest speedrank ptEF at this point.
  + This variable and the loop that is used to populate it should be deleted since it is no longer useful.
  + The function create\_FFP\_SP\_lists should no longer return this variable, and somewhere around line 530 that means the variable SRP\_index should be removed from there also.

The purpose of SRP\_EF is to find the point separating slow and negligible processes. This point is marked with SRP\_index. This may be useful information, although we never use it in the code.

* Why is this line here: SRP\_index = BP\_index

This is related to finding the separation between slow and negligible processes (see previous point). In this case, the BP is the slowest process before negligible processes are found (e.g., the slowest FQP/FFP and no slow processes or a single FRP).

* Near line 414 you have:  
   if BP\_type is None:

# Only happens if we only have FQPs since the presence of any FFP

* + I think that this should be made into an “else” statement where “if EF\_max >= tg.FFP\_floor:” occurs. If you see no objection, then change it.

This is almost right but not quite. The issue is if we have both FFPs and FQPs, then the sFFP should be the BP, but when looping over the processes we may find a non-QE process that we don’t know about yet. We have to keep scanning processes until either (1) we find the FRP or (2) we run out of processes. Only in the first case can we make an early definitive assignment of the type of BP. In the second case, since we can’t quit, if we blindly keep assigning the BP\_type after getting fast the sFFP, we might forget that we have FFPs and wrongly say that the BP is an FQP, when it really is an FFP. The way to address this is to only assign FFP as the type to FFPs while scanning the list and not assign FQP until all other possibilities have been eliminated. This is why this check happens last, after both the FRP and FFP possibilities have been ruled out.

\*\*\*

As far as I can tell, this loop:

“for i in range(BP\_index, SRP\_index + 1):”

should simply be from range 0 to len(ranked\_uEF\_list)+1

\*\*\*

This is true. However, I chose to have temporary variables as I thought they were more descriptive. In particular, I thought that len(ranked\_uEF\_list) was not obvious enough. I have kept SRP\_index + 1 but replaced BP\_index with 0.

* Why is it useful to have “set\_compression\_scheme” tied to the SP\_ranks and FFP\_ranks? Why not just allow SPs to be throttled as well? If there are no SPs, the function that tries to throttle them will be very quick. Or am I mistaken? Why is it more efficient to have these distinctions than to not have them? Because if a list length is zero, you’re not really saving any efficiency to avoid iterating over that list.

for the compression schemes, why can’t we just have unthrottled plus the general case loop?

else: # General case -- both FFPs and SPs

for i in range(0, 4):  
 for j in range(1, 4):  
 compression\_scheme.append((j, i))

also, please change from “i" and “j” to “fast\_scale” and “slow\_scale” or something like that.

These two are related. The issue is not the empty lists *per se*. It is the fact that we will have to iterate over the non-empty lists more than we would otherwise have to. For example, if we only have slow processes, we will try to throttle the non-existent fast processes three times before attempting to throttle the slow processes at the next scale. Essentially this means we would calculate the same set of slow process throttling factors three times before being able to try the next slow throttling scale. We have the same issue with only fast processes as well. Although we iterate over the fast scales first, if the target compression is not met, we will iterate over the fast scales three times before deciding that there really aren’t any more options. I have updated the counter variable names and fixed a bug in the general progression (the variables j and I were interchanged).

But I am confused about how your current code is currently handling the no compression case. It’s not obvious to me how that is being handled correctly. That would require setting the ATFs all to 1. Where does that check occur?

This happens by default. The ATF list is initialized with all ATFs equal to one in initialize\_throttling\_factors\_list. So if there is no compression or step down (step down being separate from compression), the ATF list is all ones, and no throttling will happen. If there is step down but no compression, then the ATFs for the FFPs are all set to the same value.

* ADD TO YOUR TO-DO LIST: Go over structures with Ashi of speedranked uEFs etc. to make sure they are still done in a way that he can add loops later.

The ranked\_uEF\_list structure is based around process groups (currently pairs). Each process group is a list of elementary process lists. Each elementary process list has the process number, the process EF, and a 2-tuple with indices into the unranked uEF\_list. The ranked\_uEF\_list is sorted according to the first process EF in each process group list. In paired ranking, the process groups are simply the forward and reverse processes, with the faster of the two being the lead process in the process group list (so that ranke\_uEF\_list is ranked by the faster EF in the pair).

Regarding programming concepts.

Another place this concept comes into play is “If” “else” statements. Else statements should only be used when there is genuinely a catchall desired. Otherwise an elseif should be used. Doing so will throw an error if there is a problem in the programming. If there are two cases expected, both cases should be checked for explicitly. This also makes it easy for somebody reading the code to see what the else statement is intended to cover. [In some cases like “Compression achieved” an else statement is appropriate, as long as it is commented to say that the else statement is for when compression is not achieved, but that is sortof a catch-all case anyway]

If there is an if/elif block without an else, it won’t throw an error. It will still run, it will just skip the non-existent else and continue to the next statement. It is entirely equivalent to the else/pass construct that I have used. I believe in general that I have followed this pattern of if/elif/else/pass, with the pass explicitly indicating to the reader that that case should not happen (and there is a comment stating such). If you can find specific examples where this is not the case and it’s problematic, I’ll correct them.

I am not sure what motivated you to make “old\_uEF = old\_ptEF” but I don’t see a need for this, please remove it unless you can provide a compelling reason. We just need uEF (which is the old uEF sortof by definition).

This whole loop is fairly strange in notation: for i in xrange(fSP\_index, SP\_ranks):

fSP\_index 🡨 why not call this FRP index? Am I misunderstanding what this is? I think we should just remove it, and remove the loop that assigns it. I guess this is the “first throttled slowest process” that we don’t need to keep track of?

These are related. The loop starts at the first throttled slow process and runs to the total number of slow processes (thereby skipping the unthrottled slow processes). To bootstrap the calculation of the ATFs for the slower processes, I need the uEF and ptEF of the last process. Because this process was not throttled, they are (by definition) the same. The fSP index is **not** the FRP index (which is always trivially zero). It is the index of the first process that should be throttled, based on the FRP/Nsites threshold. I don’t keep track of the first throttled slow process beyond its immediate use in this function. In looking at the code again, it does not do much, so I have combined the two loops.

if I am not mistaken, currently you have some “Case 1” “Case 2” type implicit logic that is not necessary. Sometimes that is necessary, other times it is an attempt to create more code efficiency. For those that can be removed (i.e., they wouldn’t improve efficiency by an order of magnitude anyway during a typical snapshot), remove them for the following reasons:

1. Such implicit logic is more likely to create bugs when you are programming
2. it makes it harder for somebody else to check your code or debug
3. if somebody does not realize there is implicit logic or if they do not understand the implicit logic correctly then they may create bugs when they modify the code
4. it causes the code to match the most simple written algorithm less well.

It is easier code to read if it is general code that is relatively simple, but which can pass many different test cases.

Sometimes I have early return statements if I know that subsequent code would fail or not be executed. It keeps the later code free of a bunch of ugly statements intended to prevent errors (e.g., trying to access a non-existent list element). In my view such return statements are perfectly unambiguous. If you get to that spot in the code, nothing else will happen.

I believe the only major case of ‘implicit logic’ as you call it is when we create the FFP and SP lists. This function is thoroughly tested and has not changed for a long time. I’m also not sure of a better way to write it as we have to scan the reaction list until we find the proper type for the benchmark process. This function also assembles the fast and slow lists, so we have to check every reaction. I’m also not sure I would call it implicit. The only ‘implicit’ logic I see is when we find the first completely negligible process pair, we quit scanning processes because the remaining processes are guaranteed to be negligible because the list of process pairs is sorted and we are scanning rates in descending order.

Think about line 371 for FQPs

Not sure what this means. I think it’s related to whether we want to keep FQPs and FFPs together, but I’m not sure. Unless you can give me more concrete direction, I’m not doing anything with it.

for j in xrange(len(ascend\_FFP\_list)): 🡨 please change the index to rxn rather than or rxn\_j.

There are several examples of this. How aggressive should I be in making these changes?

\* I find the "find\_process\_pairs" and tg.EF\_indices\_dict  confusing. I will probably ask for this to be changed.

Need more info.

**Outstanding**

I don't like the wording of  local\_snapshot\_idx because it is not local to that function. How about just  “snapshot\_idx” or “current\_snapshot”

If your idea is that tg.current\_snapshot += 1  would be an *absolute* counter, then it should be part of the snapshots module and maybe be called sg.total\_snapshots since that would keep track of non-throttled snapshots also.

Your print statement would then call that variable in addition to sg.kmc\_time.

I think this would be a better way of keeping track of snapshots, I think you will agree.