Data conventions / boundary rules

0.0.1 Frills

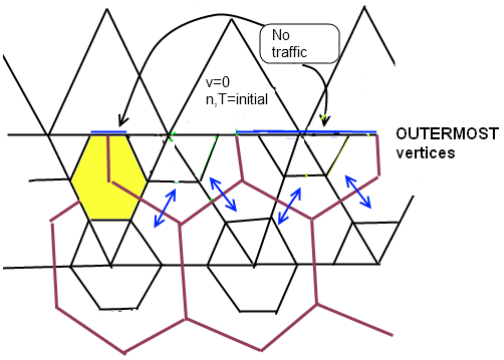
// OKay. ASK // Is there instead a way to get rid of frills. // We still want a constant number of tris per tri tile, # verts per vert tile, // we still want a geometric correspondence between tri and vert tiles. // Frills then seems inevitable.? If we knock 1 off the number of tris per tri tile, // that will bring down the number of tris by # tri tiles, which is awkward.

// ** So yes, we need frills. **

Frills are going to be along both the outermost and innermost boundaries.

- // Frill has vertex 0,1 as the two different vertices.
- // Frill has all tri neighbours set to the 1 neighbour.
- // Frill has area = 0.
- "centroid" is on boundary ? NO:
  - // So how are we going to get the corners of central cell?
  - // Do we (i) go 1/3 along tri edge or (ii) make them the average of 2 tri centroids and the vertex?
  - Way (ii) allows us not to have to access extra information than triangles in order to calculate Lap A on central cells.
  - However, if we do this then we distort central cells near the insulator.
  - And we cannot then sensibly be putting the frill centroid on the boundary. That will spoil things.
  - It is better if we do not do that then. // Put the frill centroid the other side of the boundary.
- Try to avoid traffic of nv into frills or (for now) into insulator triangles – whether by advection or diffusion.
  - Therefore we HAVE to load the flag of the triangle that we are looking into.
- When we set $\nabla^2 A$ we first set A_{frill} in a preliminary routine so that we do not need to detect frill neighbours of domain triangles.
- Ensure that in frills, $v = 0$.

0.0.2 Outermost vertex-centered cells

- Outermost cells DO allow ingress of species and DO allow ϕ to pump electrons back to the domain: this is via triangle minors v of course. There is no traffic at the outer edge of the outermost cell.
- If we choose outermost $v = 0$ – which we might as well since neither v_r nor v_z has any sense in being nonzero - then we can still evolve $\partial A/\partial t$ all the same.
- When we set $\nabla^2 A$, ∇A or $\nabla \phi$ for outermost central, we can apply boundary conditions looking outwards. Likewise setting $\nabla^2 \phi$ for outermost major. We can probably use A_{frill} to create $\nabla^2 A$.
- 

0.0.3 Insulator-crossing triangles

Parts of triangle-centered minor cells with flag CROSSING_INS will be outside the insulator and parts will be inside it.

Terminology:

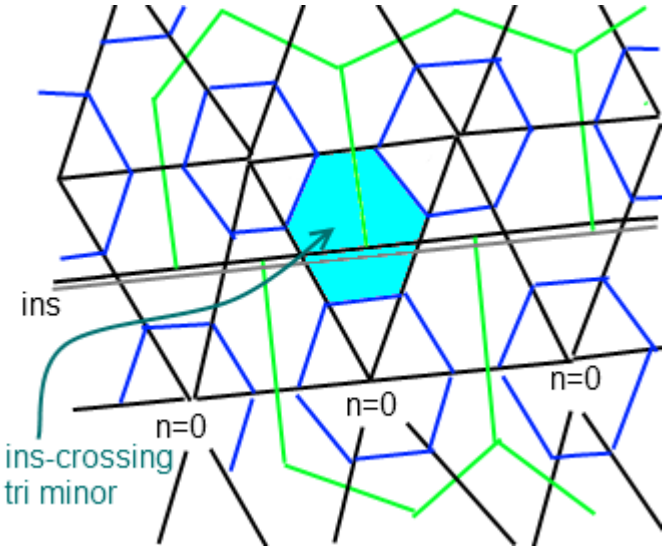
The green cells are Major Cells which store density and temperature.

The blue cells are Central Minors which store velocity.

The remainder of a triangle is a Tri Minor which store velocity.

Note that mass, momentum and heat are conserved where appropriate, but we work in v and T because they are subject to spatial smoothing. If we worked in nT we would need to do floating-point divisions every time we populate a set of shared data to get the gradient of T, so even purely from a computer efficiency perspective there was a strong argument.

The mesh continues into the insulator so that we can model A there, basically.



How to handle:

- Insulator-crossing triangles have a v that is 0 for now.
  - We do want ϕ changing at vertices to be able to push electrons out. In some insulator-crossing tris this means azimuthally pushing out. The azimuthal gradient of ϕ can be relevant, leaving $a_r = 0$. We would get the azimuthal electric and pressure gradients from where n, ϕ are stored at the vertices.
  - When we come to do device-azimuthal advection of N, NT, we can try to do this using the triangle above – though adding this branching is not ideal for CUDA so we want to avoid it.
- It IS logical to have v cells crossing a boundary so that we can decide how much traffic there is at the boundary of an electrode. But do we want this to be legislated v or evolving v?
- Insulator-crossing triangles perceive $\partial\phi/\partial r$ and $\frac{\partial}{\partial r}(nT)$ as zero, if v ever is allowed to evolve there. Allowing nonzero v means we have to create domain intersection area, which can be a little bit fiddly.
- It was nice when nv was found in major cells that abutted the insulator, for sure.
- Inner triangles should also have $v, n = 0$ so that we can do evolution of A without branching.
- Phi is defined at the vertices within the domain. Within insulator we would in 3D be able to define it, and there is a big influence from beneath.
  - However, the important thing for now is that when there is charge at the nearby vertices, they can affect v_e nearby and eject the charge.

0.0.4 Summary

- There should be now 5 possible tri flags: OUTER_FRILL, DOMAIN_TRIANGLE, INNER_FRILL, CROSSING_INS, INNER_TRIANGLE.
- whereas vertices can be OUTERMOST, DOMAIN_VERTEX, INNER_VERTEX, INNERMOST.
- whereas centrals can be OUTERMOST_CENTRAL = OUTERMOST, INNERMOST_CENTRAL = INNERMOST, DOMAIN_VERTEX, INNER_VERTEX
- Do we make different flags for OUTERMOST_CENTRAL vs such as OUTER_FRILL ???

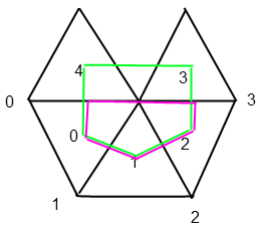
What happens for each of these, summarized for each of the 3 things: treatment of A's derivs, treatment of flows, and treatment of grad phi, nT, etc:

- In this table, "standard" means there is nothing to watch out for.

	Treatment of A derivs	Momentum flows:	Treatment of ϕ,pressure
OUTER_FRILL	A_{frill} set per BC	n/a, $nv = 0$	n/a
DOMAIN_TRIANGLE	standard	no nv into outer frill, ins tri	standard
INNER_FRILL	A_{frill} set per BC	n/a, $nv = 0$	n/a
INNER_TRIANGLE	standard	n/a, $nv = 0$	n/a
CROSSING_INS	standard	FOR NOW $nv = 0$	FOR NOW $v = a = 0$
		Mass, heat flows:	
OUTERMOST	Careful use of A_{frill}	Traffic with domain, not outwards	ϕ exists; $\nabla^2\phi$ careful; $\nabla(nT) = 0$
DOMAIN_VERTEX	standard	std: $v_r = 0$ prevents flow thru ins	Prevent Inner vertex neighs: how?
INNER_VERTEX	standard	n/a, set $n = v = T = 0$	n/a
INNERMOST	Careful use of A_{frill}	n/a, set $n = v = T = 0$	n/a

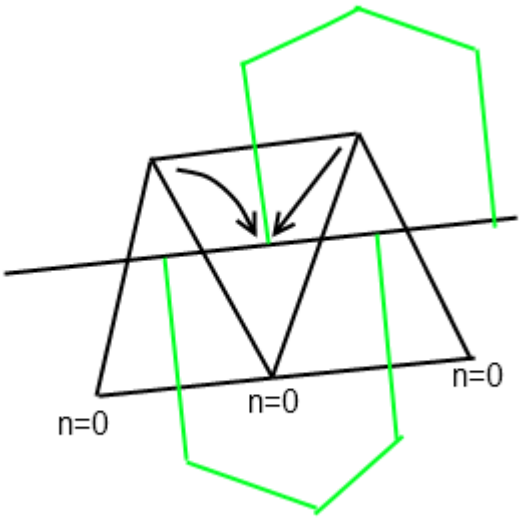
Function specifications

- Kernel_CalculateTriMinorAreas_AndCentroids [0 stack frame, 0 spill stores, 0 spill loads]
  - __shared__ f64_vec2 shared_vertex_pos[SIZE_OF_MAJOR_PER_TRI_TILE];
  - $Area = 0.5\,|(u_{2x} + u_{1x})(u_{2y} - u_{1y}) + (u_{3x} + u_{2x})(u_{3y} - u_{2y}) + (u_{1x} + u_{3x})(u_{1y} - u_{2y})|$
  - Multiply by 2/3 to roughly get tri minor area.
    - This is a KNOWN BUG, since we do not position minor nodes on tri edges.
  - char4 perinfo contains per0,1,2 and flag.
  - If vertex0 ==vertex2 on a frill tri then we get area 0 for it.
  - For insulator triangles CROSSING_INS, we want to just set this to the area intersecting the domain, and project the centroid to live on the insulator. However we don't have to implement this functionality unless we allow $v \neq 0$ in these tris. For now assume we will legislate $v = 0$ there.
- Kernel_CalculateCentralMinorAreas [0, 0, 0]
  - __shared__ f64 shared_area[SIZE_OF_TRI_TILE_FOR_MAJOR];
  - __shared__ long Indextri[MAXNEIGH_d*threadsPerTileMajor];
  - Calc area as 1/6 times sum of surrounding tri minor areas.
    - KNOWN BUG: if we position the minor nodes as average of 2 centroids and corner, then the result is not exactly 1/6 of the tri minor areas.
  - We use neigh_len not tri_len so for a frilled vertcell, we do not include area of one of the frills, IF the numbering is correct. This should not matter since frills have been assigned area 0.
- Kernel_CalculateMajorAreas [0, 0, 0]
  - __shared__ f64_vec2 shared_centroids[SIZE_OF_TRI_TILE_FOR_MAJOR]
  - __shared__ long Indextri[MAXNEIGH_d*threadsPerTileMajor];
  - __shared__ char PBCtri[MAXNEIGH_d*threadsPerTileMajor];
  - Uses $\int [\nabla_x x]\, dx$ shoelace to calculate area:
    - grad_x_integrated_x += 0.5*(unext.x+uprev.x)*(unext.y-uprev.y);
  - Uses neigh_len and tri centroids so special code is needed for OUTERMOST or INNERMOST flag. We use frill centroids. The result is the green area NOT the pink area due to not placing frill centroids at the domain boundary.



  -

- Kernel_Average_nT_to_tri_minors [0, 0, 0]
  - __shared__ nT shared_nT[SIZE_OF_MAJOR_FOR_MINOR];
  - Perform a simple average of n and T from tri corners.
  - Grabs 3 corners – for frills you will average 2/3+1/3 ; it should not be being used? I think not anyway.
    - $\nabla(nT)$ at outermost should not depend on frill n, T values, if it is taken at all.
  - In case of CROSSING_INS we only add n, T where $n > 0$. We ASSUME we shall keep n=0 for all out-of-domain vertices.



  -
  - No periodic data needed.
  - In future this routine very likely needs to be a lot more complex. It is probably not adequate to assume simple averaging once there is a front of electrons. This is the n_{tri} that is being used for boundary flows – we should make a model on each shard of each major cell, and the upwind shard is then the relevant one.

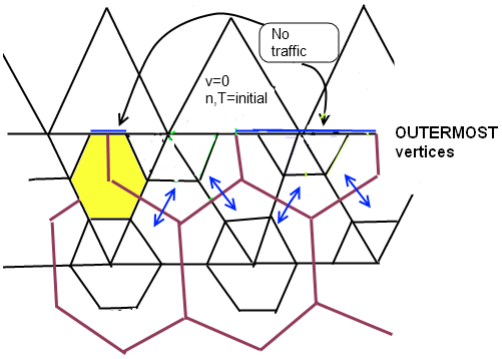
- Kernel_GetZCurrent [0, 0, 0]
  - ((minor_info.flag == DOMAIN_MINOR) || (minor_info.flag == OUTERMOST_CENTRAL))
  - So insulator-crossing tris are being ignored.

- Kernel_Create_v_overall_and_newpos [0, 0, 0]
  - Runs for DOMAIN_VERTEX. Put 0 otherwise, including for OUTERMOST.
  -

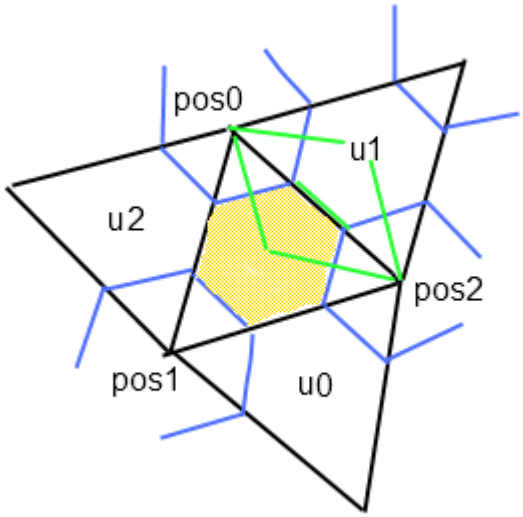
$$v_{overall} = \frac{m_n n_n v_n + m_{ion} n_{ion} v_{ion} + m_e n_e v_e}{m_n n_n + m_{ion} n_{ion} + m_e n_e}$$

- Kernel_Average_v_overall_to_tris: [0, 0, 0]
  - __shared__ f64_vec3 shared_v[SIZE_OF_MAJOR_FOR_MINOR];
  - ((perinfo.flag == DOMAIN_TRIANGLE) || (perinfo.flag == CROSSING_INS))
    - All other cases $v_{overall} = 0$.
  - uses perinfo.per0,per1,per2 to handle PB rotation.
  - Perform simple average from corners.
  - For CROSSING_INS, we set the radial component to zero. However we still took an average of 3 corners. At best, this means we averaged with 0 from inner vertex.

- Kernel_Average_nnionrec_to_tris: [0, 0, 0]
  - __shared__ nn shared_nn[SIZE_OF_MAJOR_PER_TRI_TILE];
  - ((perinfo.flag == DOMAIN_TRIANGLE) || (perinfo.flag == CROSSING_INS))
    - All other cases $n_{ionise} = 0$.
  - nn_out.n_ionise = THIRD*(nn0.n_ionise+nn1.n_ionise+nn2.n_ionise);

- Kernel_RelAdvect_nT [72, 120, 160]
  - __shared__ f64_vec2 p_tri_centroid[SIZE_OF_TRI_TILE_FOR_MAJOR];
  - __shared__ f64_vec2 p_nv_shared[SIZE_OF_TRI_TILE_FOR_MAJOR];
  - __shared__ f64 p_T_shared[SIZE_OF_TRI_TILE_FOR_MAJOR];
  - __shared__ long Indextri[MAXNEIGH_d*threadsPerTileMajor];
  - Each species is done in sequence, retaining the position data etc.
  - if ((info.flag == DOMAIN_VERTEX) || (info.flag == OUTERMOST)) ->
    - because we do allow traffic in/out of Outermost;
    - Make sure no traffic at the back of Outermost. - tick
    - Make sure no traffic through insulator. As long as $v_r = 0$ in tris, no problem.
  - This is to avoid a situation where a domain vertex has to test whether it is looking into an OUTERMOST neighbour.
  - 

  - We use info.neigh_len but we are dealing with triangle pairs to look at edges for advection. Therefore at the outermost edge, we need the numbering where this will exclude the "edge" between two frills.
  - Uses **info.has_periodic** for PB rotation of tri nv, nvT data.
  - Accumulate:
    - f64 flow = 0.5*h*((nv1+nv2).dot(edgenormal));
    - mass -= flow;
    - flow = 0.5*h*((nvT1+nvT2).dot(edgenormal));
    - heat -= flow;
  - mass += nTsrc.n*area_old; heat += nTsrc.n*nTsrc.T*area_old;
  - nT_out.n = mass/area_new; nT_out.T = heat/mass;
  - Compressive: $T_{out} = T_{out}\left(1 - \frac{2}{3}\frac{(n_{out}-n_{src})}{n_{src}} - \frac{1}{9}\frac{(n_{out}-n_{src})^2}{n_{src}^2}\right)$
- Kernel_Populate_A_frill
  - Detect where flag == OUTER_FRILL || flag == INNER_FRILL ;
  - Look domain-inwards to neighbour and populate accordingly.
- Kernel_Compute_Grad_A_minor_antiadvect: [144, 480, 748] – not great. Note (144-32)/8=14
  - __shared__ f64_vec3 A_tri[threadsPerTileMinor];
  - __shared__ f64_vec2 tri_centroid[threadsPerTileMinor];
  - __shared__ f64_vec3 A_vert[SIZE_OF_MAJOR_PER_TRI_TILE];
  - __shared__ f64_vec2 vertex_pos[SIZE_OF_MAJOR_PER_TRI_TILE];
  - __shared__ long IndexTri[SIZE_OF_MAJOR_PER_TRI_TILE*MAXNEIGH_d];



–
– We move around taking 6 quadrilaterals.

$$\int \nabla A_y + = \left(\frac{1}{12} \left(A_{prev,y} + A_{next,y} \right) + \frac{5}{12} \left(A_{0y} + A_{out,y} \right) \right) \perp$$

where $\perp$ is 90 degrees from the green edge line shown on the figure.

– Periodic rotation flags for corners and for neighbour centroids come from two CHAR4 objects that are loaded.

– Looking out into frills we have to apply Boundary Conditions:

  * Set A_{frill} beforehand and it solves a lot of problems.

– if ((perinfo.flag == DOMAIN_TRIANGLE) || (perinfo.flag == CROSSING_INS))

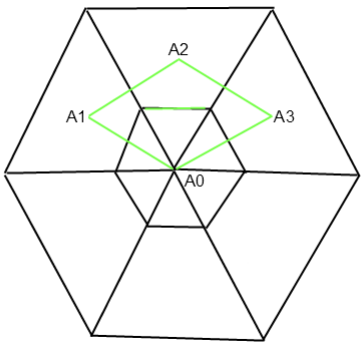
  * We did not apply this test to most of the code though!!! Try both ways.

– =————————————————-

– For half the threads, we carry on to then have a go at the central grad A. [Note that even if we comment this part, we do not dramatically reduce the overspend on registers and local memory.]

– Periodic state for triangle is used to infer whether tri centroid and A need to be rotated.

– For OUTERMOST, we will not be moving the vertex so can simply write if (info.flag == DOMAIN_VERTEX)



–

– Use $\frac{1}{12}(A_1 + A_3) + \frac{5}{12}(A_0 + A_2)$ as the average of A on the edge of the central polygon.

– p_A_out[BEGINNING_OF_CENTRAL + index].z += anti_Advect.z = h*v_overall.dot(gradAz);

- Kernel_Compute_Lap_A_and_Grad_A_to_get_B_on_all_minor: [216, 1536, 2024] – terrible?

  – __shared__ f64_vec3 A_tri[threadsPerTileMinor];

  – __shared__ f64_vec2 tri_centroid[threadsPerTileMinor];

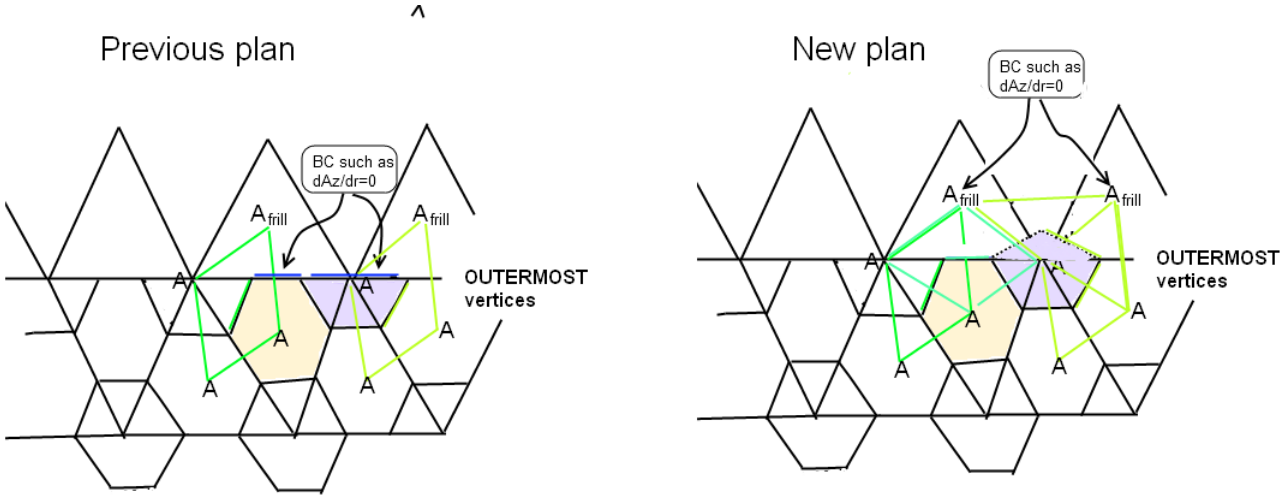
  – __shared__ f64_vec3 A_vert[SIZE_OF_MAJOR_PER_TRI_TILE];

  – __shared__ f64_vec2 vertex_pos[SIZE_OF_MAJOR_PER_TRI_TILE];

  – __shared__ short shared_per[threadsPerTileMinor];

  – __shared__ long IndexTri[SIZE_OF_MAJOR_PER_TRI_TILE*MAXNEIGH_d];

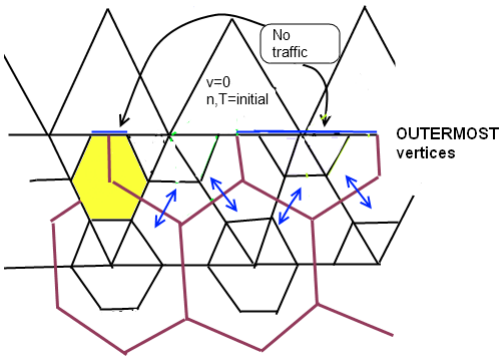
- Similar routine to Grad A antiadvect. For some reason comes out much worse for local memory footprint, even if you remove B.
  - coeff = ((u2.y-u0.y)*edgenormal.x + (u0.x-u2.x)*edgenormal.y)/shoelace;
  - LapA.z += coeff*(A0.z-A_out.z); // A_1~pos0 A_2~pos1
  - coeff = ((pos1.y-ourpos.y)*edgenormal.x + (ourpos.x-pos1.x)*edgenormal.y)/shoelace;
  - LapA.z += coeff*(A_1.z-A_2.z);
  - B.x += (TWELTH*(A_1.z+A_2.z)+FIVETWELTHS*(A0.z+A_out.z))*edgenormal.y;
  - B.y += -(TWELTH*(A_1.z+A_2.z)+FIVETWELTHS*(A0.z+A_out.z))*edgenormal.x;
- At the outer and inner edge of memory, we apply BC's to create Lap A since this affects whether waves of A are flowing in and out.



-
- We have to set A_{frill} according to the BC before we call this kernel. In order to do that we want to have some idea which tiles contain frill triangles.
- It seems best for outermost central that we allow A to evolve with Lap A but in order to do that, we assume that we take a trapezoid and the BC is applied along the outer edge of the trapezoid.
  - Does this mean we should let J be defined at outermost vertices? We should assume that $J = 0$ there. Assume $v = 0$.
  - For A_{xy} the BC is not that the outer edge is ignored. It is something like that A_{xy} decreases as $\frac{1}{r}$, and at the innermost edge decreases inward as if $A_{xy}/r =$ constant.
- At insulator, A is treated the same way as everywhere else. It is defined inside and outside the insulator and we can work out B and Lap A at the insulator tri centroids.
  - What about evolving $\frac{dA}{dt}$ there? It would not be wrong to treat in a weak way: allow that dA/dt is affected according to the proportion of the triangle that lies within domain. Let v be defined nonzero ($v_r = 0$) at the insulator.

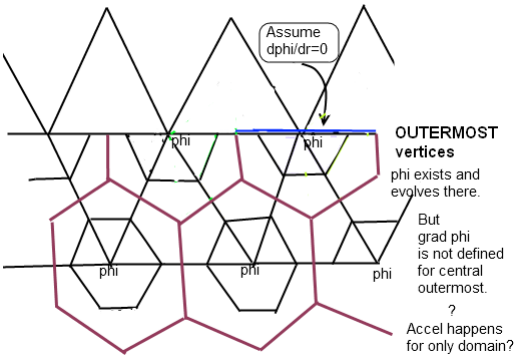
- Kernel_Rel_advect_v_tris : [128, 172, 280]
  - __shared__ f64_vec2 tri_centroid[threadsPerTileMinor];
  - __shared__ f64_vec3 v_tri[threadsPerTileMinor];
  - __shared__ f64_vec2 n_vrel_tri[threadsPerTileMinor];
  - __shared__ f64_vec2 n_vrel_central[SIZE_OF_MAJOR_PER_TRI_TILE];
  - __shared__ f64_vec3 v_central[SIZE_OF_MAJOR_PER_TRI_TILE];
  - __shared__ f64_vec2 vertex_pos[SIZE_OF_MAJOR_PER_TRI_TILE];
  - nvrel.x = n_own*(v_own.x - v_overall.x);
  - if (perinfo.flag == DOMAIN_TRIANGLE).
    - 4 possible tri flags: OUTER_FRILL, DOMAIN_TRIANGLE, INNER_FRILL, CROSSING_INS.
    - whereas vertices can be OUTERMOST, DOMAIN_VERTEX, INNER_VERTEX, INNERMOST.
  - Periodic handling is peculiar: CHAR4 perneigh = p_tri_per_neigh[index]; which stores whether neighbour triangle is relatively rotated.
  - Boundaries:

    * Do not flow momentum out into a frill ... repeat diagram:



    *
    * Do not flow momentum through the insulator: test for 2 CROSSING_INS triangles. Should that be unnecessary if both have $v_r = 0$? Check that the flow will come out zero.
    * SO FAR NO EFFORT MADE to actually prevent momentum from disappearing these ways.

- Kernel_Rel_advect_v_central : [56, 52, 56]
  - __shared__ f64_vec2 tri_centroid[SIZE_OF_TRI_TILE_FOR_MAJOR];
  - __shared__ f64_vec3 v_tri[SIZE_OF_TRI_TILE_FOR_MAJOR];
  - __shared__ f64_vec2 n_vrel_tri[SIZE_OF_TRI_TILE_FOR_MAJOR];
  - __shared__ long IndexTri[threadsPerTileMajor*MAXNEIGH_d];
  - __shared__ char PBCtri[threadsPerTileMajor*MAXNEIGH_d];
  - PBCtri being preferred to having shared_per for each triangle.
  - (info.flags == DOMAIN_VERTEX)
- Kernel_Compute_grad_phi_Te_centrals : [0, 0, 0]
  - __shared__ f64 p_phi_shared[threadsPerTileMajor];
  - __shared__ f64 p_Te_shared[threadsPerTileMajor];
  - __shared__ f64_vec2 p_vertex_pos_shared[threadsPerTileMajor];
  - __shared__ long indexneigh[MAXNEIGH_d*threadsPerTileMajor];
  - (info.flag == DOMAIN_VERTEX)
- Kernel_GetThermalPressureCentrals : [0, 0, 0]
  - __shared__ f64 p_nT_shared[threadsPerTileMajor];
  - __shared__ f64_vec2 p_vertex_pos_shared[threadsPerTileMajor];
  - __shared__ long indexneigh[MAXNEIGH_d*threadsPerTileMajor];
  - (info.flag == DOMAIN_VERTEX)
- Kernel_Compute_grad_phi_Te_tris [0, 0, 0]
  - __shared__ f64 p_phi_shared[SIZE_OF_MAJOR_PER_TRI_TILE];
  - __shared__ f64 p_Te_shared[SIZE_OF_MAJOR_PER_TRI_TILE];
  - __shared__ f64_vec2 p_vertex_pos_shared[SIZE_OF_MAJOR_PER_TRI_TILE];
  - (tri_info.flag == DOMAIN_TRIANGLE)
- Get_Lap_phi_on_major [0, 0, 0]
  - __shared__ f64 p_phi_shared[threadsPerTileMajor];
  - __shared__ f64_vec2 p_vertex_pos_shared[threadsPerTileMajor];
  - __shared__ long Indexneigh[MAXNEIGH_d*threadsPerTileMajor];
  - __shared__ char PBCneigh[MAXNEIGH_d*threadsPerTileMajor];
  - __shared__ f64_vec2 tri_centroid[SIZE_OF_TRI_TILE_FOR_MAJOR];



  - KNOWN BUG: It is not getting it right in case that the major cell abuts the insulator, because it is assuming that every centroid is at 1/3 simple average.

- Kernel_GetThermalPressureTris [0, 0, 0]
  - __shared__ f64 p_nT_shared[SIZE_OF_MAJOR_PER_TRI_TILE];
  - __shared__ f64_vec2 p_vertex_pos_shared[SIZE_OF_MAJOR_PER_TRI_TILE];
  - Multiple times: (tri_info.flag == DOMAIN_TRIANGLE)
- Kernel_Advance_Antiadvect_phi,phidot [0, 0, 0]
- Kernel_Antiadvect_A_allminor [0, 0, 0] - takes grad A as input

---

- . Advance $\frac{\partial \phi}{\partial t}$ to half-time using $\nabla^2 \phi_k$ and advance ϕ to half-time using that updated $\frac{\partial \phi}{\partial t}$. Remember we do not improve the advance method to a corrector in this version.
- . Advance A to half-time using $\frac{\partial A}{\partial t}_k$. A is trapezoidal advanced, ϕ is midpoint but we inconsistently do a fwd Euler step of it to create $\nabla^2 \phi_{1/2}$.
- . Compute grad phi 1/2, Lap A 1/2, B 1/2
- . Calculate half-time minor areas and estimated densities. (We do need densities now...)
- . Do some species advection. Compressive heating. This is both for major and for minor cells!
- . Calculate thermal pressures at half time. This is for all minor cells, where we want to do acceleration.
- . Midpoint step (feint) applies on all minor cells. Here we use shared mem only for . We can choose to synchronize threads only after submitting both types of routines.
- IS IT POSSIBLE to make an array that contains both the triangle minors and the others? NOT in general that useful?? Well it is though when you come to the midpoint routine. Let's be very careful here. Maybe there IS a way.
  - Problem with this - to update nv, we need to be taking grad ϕ on triangles. Then we have a thread per triangle, and store
  - Better to make one list but not contiguous. There is the same data but triangle minors are contiguous and central minors are contiguous list.
  - To get Lap A, on edge minor, we are looking at the data from 3 centrals and 3 edge minors. How do we get at that? Load in 2 lists - the minor tile and the major tile corresponding centrals.
  - To get v for v_overall at vertices, load from centrals list. Save to corresp list, then it is gathered by each triangle.
  - Data structure: minor {nv, area} ; major {n,nT,} (wtf not nT if we use nv... such a sucker for changing things up though. My midpt equation is written for v_e.).
  - When we collect current, run threads for ALL minors.

1. TO DO NOW: Have a look and note down trap vs midpoint in the scheme:
   (a) A is trapezoidal, ϕ midpoint. $\frac{\partial \phi}{\partial t}$ trapezoidal.
   (b) But using explicit ticked-forward ϕ towards $\frac{\partial \phi}{\partial t}$. There may be a more sensible way than that?
      i. For a more advanced method, try something approaching a backward-ticked $\nabla^2 \phi_{1/2}$ towards $\frac{\partial \phi}{\partial t}$. That makes an ODE though ... maybe something like

$$\phi_{1/2} = \phi + \frac{h}{2}\left(\frac{\partial \phi}{\partial t}_k + \frac{h}{2} c^2 \left(\nabla^2 \phi_k + 4\pi \rho_{1/2}[?]\right)\right)$$

      – at any rate mix it up a bit, because midpoint ϕ does not feel like the halfway one should be an Euler step from ϕ_k.

2. TO DO NOW: Have a look and ask, would it be easy to rewrite the midpoint advance for $n_s v_s$ instead of v_s?

The matter of $\{n, v\}$vs nv ==========-
nv can be better/worse because

1. It is conserved. <— but we know how to conserve mom under advection if using n, v, T.
2. We can still create grad v on the fly, if we have n's estimate stored. <— yes but it requires a division.
3. We use n's estimate to decide the effect of grad phi ... equivalent with multiplying against estimate when we advect the electrons. NOTE that conductivity is only weakly dependent on n because of the cancellation with n in ν — that WOULD be a strong argument if our timestep was long enough for much resistivity to apply, and is perhaps a good argument as regards what actual current ought to flow. <—— but in reality not strong argument
4. It is what is used for the advection. <—— true ... but loading 1 extra value we can recreate nv there.
5. We can directly add due to thermal pressure .... but with v we directly add due to grad ϕ.
6. nv seems like the more relevant variable than v: we are playing with current and charge, more than anything. (This is true but not material.)
7. We need to do relative advection $n_s\left(v_s - v_{overall}\right)$ which is difficult with nv.
8. Conclusion is to use v as variable. We can try hard to add the correct amount of momentum when mom is changing.

Offsets create an issue for 3D. Let's consider a couple of things we could do.

vxy in the plane has to be in a tube itself. We need to advect and diffuse vxy vertically.

If we just stuff v_xyz into completely offset cells — above triangles, here is a problem, the same triangles do not even exist in the next mesh sometimes.

What then do we do?

=

Why do we insist on tubes at all? Why not prisms that meet a number of other prisms in some messy way?

Tetrahedralization somehow determines what is going to be connected.

Somehow some intelligent way of tearing when we get too far from a linkup making sense.

Chop off corners of tetrahedron –> truncated tetrahedron. It will have hexagonal and triangular sides.

=

There probably are ways. I clearly want to deal with offset v in 2D so we should deal with this. Probably we do want the elegant 3D way that puts v_xyz on the same place. Presently edge lines intersect 2 minor cells. We would find that edge faces of the major cells would intersect multiple minor cells.

Here is a thought. We should try to align the connections between the planes according to which direction current is flowing. We would like there to be faces transverse to the current flow when possible. Then we can even try to reorient the planes gradually so that this is becoming transverse to the plane also.

=========

BACK TO 2D —