1 Explanation of Boundary Fluxes

To compute the boundary fluxes, we use a similar methodology as was implemented for the virtual radiometer model, and the 6 Flux method. Namely, we generate rays over a hemisphere, rotate the direction of the ray into the appropriate hemisphere for the boundary at hand, trace rays as usual, then weight the rays by the cosine of the polar angle from the surface normal. The details of each of these steps are given below.

1.1 Generating rays on a hemisphere.

The full details of how to generate randomly distributed rays on a hemisphere is given in the virtual radiometer section (!! virtual radiometer section). Recall that the because a hemisphere is symmetric about the surface norml, that for the azimuthal, ϕ, we can simply use $2\pi R_1$, where R_1 is a random number between 0 and 1, and therefore achieve the appropriate range of ϕ of 0 to 2π. For the polar angle, θ, because the area of a given ring of the hemisphere is a function of the polar angle, we must scale our random number by the arccosine in order to achieve equidistribution of rays throughout the solid angle. Therefore,

$$\theta = acos(R_2).$$

1.2 Rotating the rays into the appropriate hemisphere

When a ray direction has been selected, initially, the ray will be in oriented in the positive z direction, as if it were originating from the top face of a cell. This direction must be adjusted to lie within the appropriate hemisphere for the face at hand. For a structured Cartesian mesh, all of the surface normals of the cells are aligned in the coordinate directions. This greatly simplifies the rotation of the rays as it negates the necessity of using a rotation matrix, as was done for virtual radiometers with arbitrary orientations. To re-orient a ray into the appropriate direction, a simple rearrangement of the vector indices is implemented. This adjustment takes place as follows, where face is an enumeration with the following order: E,W,N,S,T,B. Notice that this enumeration is slightly different than the face enumeration that is passed in from a call to the Uintah type "face" iterator, which has the order: W,E,S,N,B,T. A simple array called RayFace, with values [1,0,3,2,5,4] can be used to ameloriate the problem, as the RayFace[Uintah face] will return the proper faces. With the proper face enumeration, the direction is reassigned onto the face at hand, and the sign of one of the components may be reversed as well, if the current face is E,N, or T, as shown in Tab. (1). Numerically, this appears as

```
Vector tmpry = directionVector;
directionVector[0] = tmpry[indexOrder[0]] * signOrder[0];
directionVector[1] = tmpry[indexOrder[1]] * signOrder[1];
directionVector[2] = tmpry[indexOrder[2]] * signOrder[2];
```

One may note that for any face, the ray direction will always point toward the inside of the cell, placing the first segment length of a ray through the origin cell. This is beause the operation that loops through the cells in the domain to identify which cells have boundary faces, loops through the interior cells. One could imagine a scenario where the algorithm would loop through the "extra" or boundary cells and identify which of those have faces that are adjacent to the flow cells. In this scenario, several modifications to the algorithm would be necessary. First, the positive and negative faces would need to be reversed, as a west boundary face would need to have rays placed on its east face in order to determine the flux at the actual interface between flow cells and boundary cells. Second, the hemisphere would then be on the outside of the cell face as opposed to the inside, as rays should not be traced through boundary material. This would lead to the third adjustment that would need to be made, which affects the intensity solver of the ray tracer. Namely, the first cell being referenced for temperature and absorption coefficient would need to be on a lag, so as to not reference the origin cell for the first segment length, since the ray would not pass through the origin cell at all, but would begin at its face and continue outward. For these reasons, I have chosen to loop through the interior cells to find those with boundary faces as opposed to looping through exterior cells.

1.3 Shifting the rays onto the appropriate face

Similar to adjusting the ray location from a default hemisphere, points on a plane are generated on a default surface, and therefore require adjustment onto the proper face. By default, points that represent the ray origins are generated on the S face (see Fig. (1)), which can then be moved onto the appropriate face by reordering the indices and applying a shift value if the face of interest is E,N, or T. This method holds for non-cubic cells as well, given that the unity shift value is scaled by the ratio of Dy to Dx for the y direction, and Dz to Dx for the z direction. Numerically, this appears as

```
Vector tmpry = location;
location[0] = tmpry[indexOrder[0]] + shift[0];
location[1] = tmpry[indexOrder[1]] + shift[1] * DyDxRatio;
location[2] = tmpry[indexOrder[2]] + shift[2] * DzDxRatio;
```

1.4 Ray tracing and weighting of rays

Once a location and direction have been specified for a given ray, ray marching, and the update of intensity is handled in the same manner as is done for the flux divergence solver, and the virtual radiometer solver. To avoid code redundancy, the ray marching and intensity solver has been abstracted into its own method called "updateSumI." Because this method returns a running total of intensity for a given cell, and because the boundary flux solver doesn't weight rays equally, the intensity of each ray must be known. To allow for this, the current total of intensity is subtracted from the previous total intensity, to yield a unique intensity for the ray, which can then be weighted by the cosine of the polar

face	new direction index order
0	2,1,0
1	2,1,0
2	0,2,1
3	0,2,1
4	0,1,2
5	0,1,2
face	new direction sign
0	-1,1,1
1	1,1,1
2	1,-1,1
3	1,1,1
4	1,1,-1
5	1,1,1
face	new location index order
0	1,0,2
1	1,0,2
2	0,1,2
3	0,1,2
4	0,2,1
5	0,2,1
face	new location shift
0	1,0,0
1	0,0,0
2	0,1,0
3	0,0,0
4	0,0,1
5	0,0,0

Table 1: Reordering of indices for adjustment of ray direction and origin location as a function of cell face. Also shown are the values that allow for location shift and direction sign change.

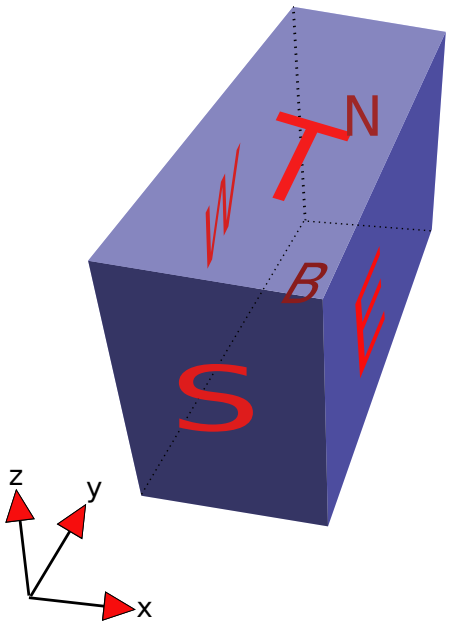

Figure 1: A hexahedron with its 6 faces labeled.

angle, to give the flux contribution from that ray. The flux from all rays can then be summed and weighted by the solid angle that each subtends, to yield an incident flux for the face at hand, as follows

$$q = \frac{N}{2\pi} \sum_{ir=1}^{N} I_i(ir) cos(\theta(ir)),$$

where $I_i(ir)$ and $\theta(ir)$ are the incident intensity and polar angle, respectively, for a given ray, and $\frac{N}{2\pi}$ is the solid angle that each ray subtends. Notice that the solid angle is assumed constant, given the equi-distribution for a large number of rays, and is therefore removed from the summation, improving numerical efficiency.

1.5 Ray Convergence Analysis

Verification testing was performed on the boundary flux calcuations. The benchmark case is the Burns and Christon case which has been used in prior verification for the flux divergence results, but also contains flux results for the same cubic, trilinear case with cold black walls. Agreement between our computed results and the Burns converged results was obtained. An increase in the number of rays led to a decrease in the L1 error norm at the expected convergence of $\frac{1}{2}$ order See Figs (2,345).

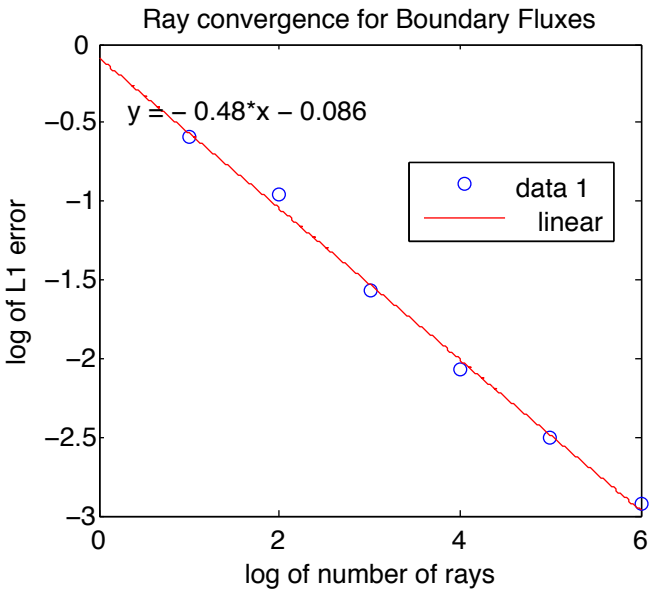

Figure 2: Ray convergence for Boundary Fluxes.

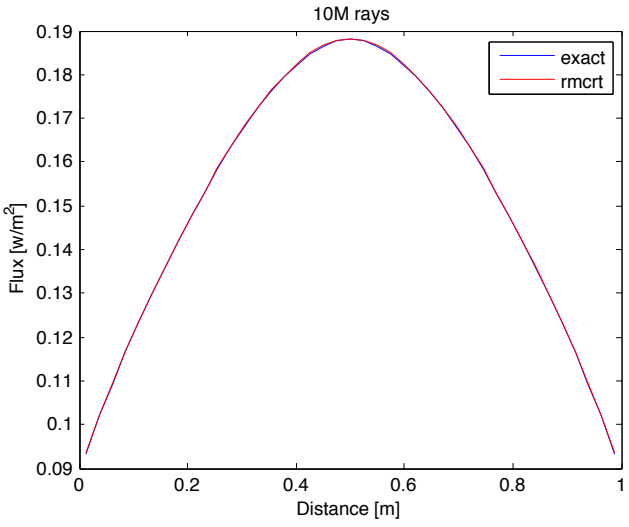

Figure 3: RMCRT vs. Burns' converged solution at 10M rays.

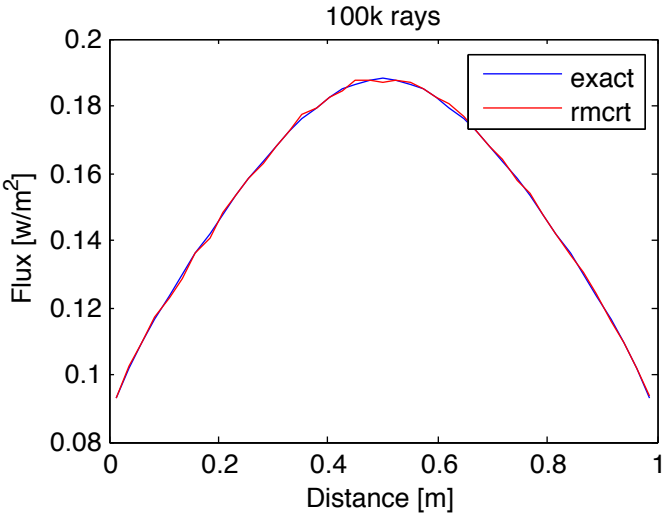

Figure 4: RMCRT vs. Burns' converged solution at 100k rays.

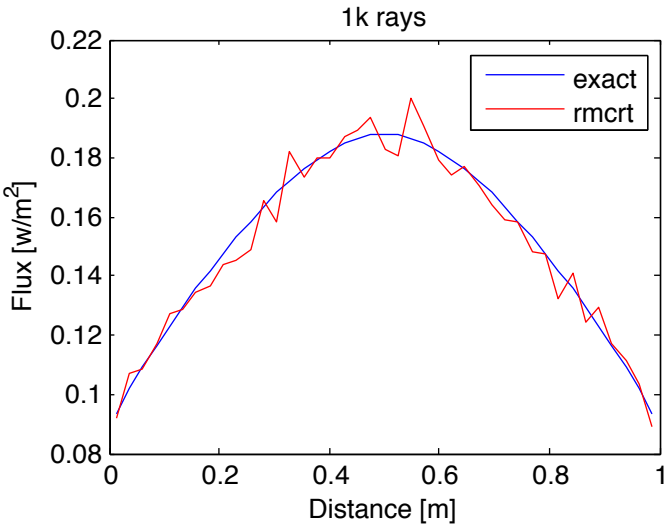

Figure 5: RMCRT vs. Burns' converged solution at 1,000 rays.

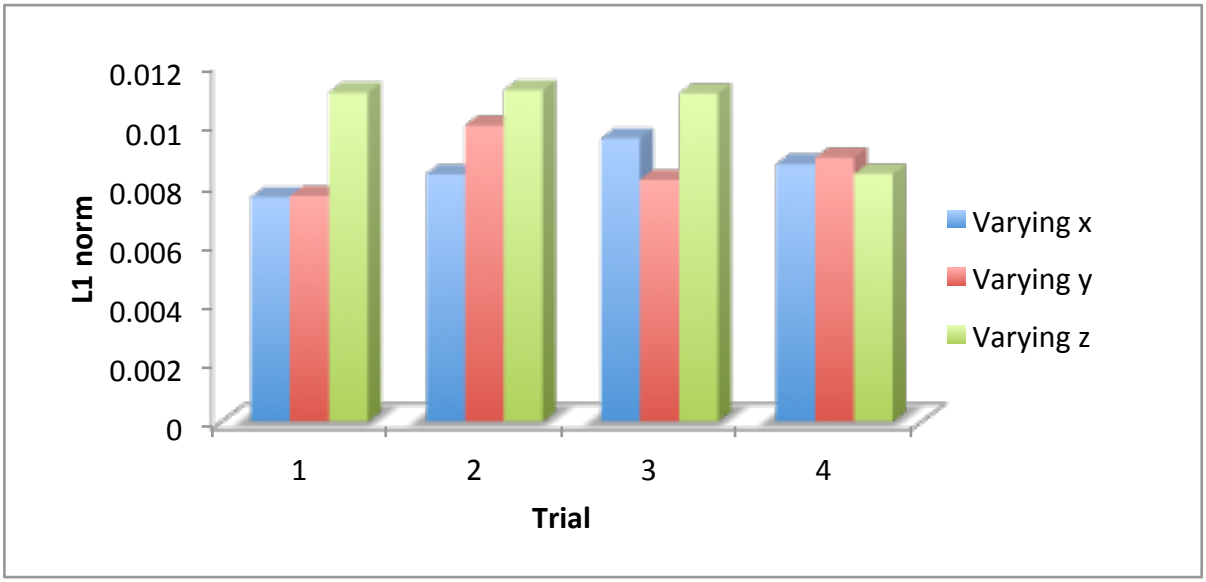

Figure 6: Invariability of the L1 error norm of the fluxes as a function of direction for the symmetric Burns and Christon case.

To ensure the code did not have a bias in one or more of the Cartesian directions, the L1 error norms from the 2 center lines of each of the 6 faces of the unit cube of the Burns case were analyzed. Four of these twelve center lines are found by varying the x values from zero to one. Similarly, there are 4 lines in the y, and z directions. Figure (6) demonstrates the lack of a bias, and therefore invariability of the L1 error norm as a function of direction, which is as expected for this symmetric case.

1.6 Storage

The Uintah framework is set up in such a way that the variable types that are used for storage in the data warehouse allocate memory sufficient to store a value for each cell in the domain. Because not all cells in the domain will contain a value for a boundary flux, this technique ties up memory that is never used. To avoid wasting memory, std::map variables were implemented where values for the boundary flux exist only for cells that contain a boundary, such as walls and intrusion. This map has been labeled "cellToValuesMap." The key to the map is a std::vector comprised of the cell index and enumerated face value, and the mapped value is the flux. To handle the complexity of multiple patches in a domain, a larger map was created that houses the patchID as the key, and the corresponding cellToValuesMap as the value. The time required to create these maps is comparable to the time required to loop through the cells in a patch and assign only a face value. Similarly, the time required to reference

	std::map	CCVar
Create	0.22 sec	0.14 sec
Run	212.22 sec	212.29 sec

Table 2: Time comparison to create and run std::map vs Uintah's CC Variables.

the map and solve for the fluxes on these mapped surfaces is comparable to the time required to solve the fluxes using CCVariables. These results are shown in Tab. (2)