Behavior of Molten Weld Pool in TIG Welding (2-D)

M475 - Spring 2018
Final Draft
https://github.com/rekeshali/weld-pool
05/01/18

Rekesh Ali, Vivek Chawla, Eric Heikkenen

1 Introduction

This study investigates the behavior of of a molten weld pool in a tungsten-inert-gas (TIG) welding process. Welding metal is a difficult procedure because there are many different types of metals and tool settings one can choose for a job. Because of this, welds can be unsatisfactory if the right choices are not made. The objective here is to provide a baseline set of information for a variety of unfamiliar metals and surface conditions. This focus is of interest to many parties because it can elucidate on the size of weld pools and therefore joints, the energy required to make satisfactory joints, and the behavior of certain metals.

The problem is modeled by conserving enthalpy within a 2-D cross sectional area. The problem is solved using an explicit numerical finite difference scheme. The computer model is capable of surface temperature and flux boundary conditions. Phase fronts and temperatures are extracted from the numerical solution. The produced results have constant density throughout the mixture, and phase based specific heat and thermal conductivity.

2 Problem Description

In TIG welding we have a metal that undergoes extreme arc heating at the surface. The heat will propagate from the surface out to cooler areas along the path of least resistance, which is typically within the metal itself. The influx of heat can be greater than the mass is properly able to diffuse, and so some parts closer to the surface being heated will store great amounts of internal energy. Eventually, the total energy within the area adjacent to the surface will reach a critical point that denotes the change of phase between solid and liquid. The melting phenomena does not happen all at once but creates a mushy zone where the area is a fraction of both solid and liquid. This mushy zone delineates the boundary between the two phases. The mushy zones will eventually become fully liquid and the boundary will grow outwards as the heating continues. The result is a liquid pool confined to the surface of a solid material.

In this project, a computational analysis is done using a mathematical model to get the temperature profile and to identify the location and volume occupied by the phase front for a metal undergoing TIG welding at any time $t > 0$. One end of the metal is exposed to the welding electrode and the other end ideally insulated with adiabatic walls. The initial temperature of the metal slab is constant and below the melting point of the metal. The mathematical model used is Enthalpy method. The problem is modelled in a very simple manner. A metal slab $0 < x < L$ undergoing TIG welding is modelled in a one-dimensional manner i.e. it is assumed that there is no variation in the properties across the cross-section. Hence a differential control volume becomes the product of cross-sectional area and differential distance. It is assumed that the mode of heat transfer to the metal slab from the welding electrode is via convection. It is assumed that the specific heat in liquid and solid state remains constant as the temperature varies.

Hence the mathematical problem to solve is the energy conservation problem,

$$\int_{t_n}^{t_n+1} \frac{\partial}{\partial t}\left(A\int_{x_{j-1/2}}^{x_{j+1/2}} E(x,t)\mathrm{d}x\right)\mathrm{d}t = -\int_{t_n}^{t_n+1} A\int_{x_{j-1/2}}^{x_{j+1/2}} q_x(x,t)\mathrm{d}x\mathrm{d}t$$

Where,
E is the Enthalpy,
q is the flux
The Enthalpy equation assuming specific heat capacity in solid and liquid phase to be constant,

$$E = \begin{cases} \rho c_S(T - T_m), & T < T_m \\ \rho c_L(T - T_m) + \rho L, & T > T_m \end{cases} \tag{1}$$

Where,
ρ is the density,
c_s, c_L are the specific heat capacities in solid and liquid phase respectively.
L is the latent heat
T is the temperature
T_m is the melt point temperature
The flux equation is given by,

$$q = -kT_x \tag{2}$$

Where,
k is the diffusivity,
T_x is the temperature gradient.
The initial condition is given by,

$$T(x,0) = T_{init}(x), \qquad 0 < x < l \tag{3}$$

The boundary conditions are given by,

$$q(0,t) = h[T_\infty(t) - T(0,t)], \qquad t < 0, \tag{4}$$

$$q(L,t) = 0, \qquad t > 0 \tag{5}$$

The aim is to solve equations (1)-(3) using the initial and boundary conditions (4)-(6). The goal is to get the temperature at any point $x, (0 < x < l)$, at any time $t > 0$. At every time step, enthalpy is used to calculate the location and the distribution of the flow front.

This location and density of flow front is identified using the ratio of Enthalpy of a control volume $dx * A$ with the Latent heat at any time t greater than 0.

$$\begin{aligned} E_j^n &\leq 0 && \text{(solid)} \\ 0 < E_j^n &< \rho L && \text{(interface)} \\ E_j^n &\geq \rho L && \text{(liquid)} \end{aligned} \tag{6}$$

3 Methodology

The goal of the solution is to be able to identify where the phase front is and what the temperatures are in the material. Many methods, known as front tracking schemes, try to solve for the fronts explicitly. This can be quite cumbersome, due to the inherent geometric non-linearity. That said, an enthalpy method does not require explicit phase tracking, phases are extracted from the solution itself.

We will examine the problem above numerically for a one dimensional slab of material spanning $0 < x < l$ for a time $t \geq 0$. The space is split into a finite number of segments containing M nodes where each node is denoted by x_j and the left and right boundaries exist at x_0 and x_{M+1}, respectively. The boundaries between nodes are denoted as $x_{j\pm1/2}$ where plus is the right boundary of the j^{th} volume and minus corresponds to the left. The solution is specified at any time t_n and time dependence of a value is denoted with a superscript n. For the purposes of this study, the mesh will be uniform and time step constant throughout; the density will remain the same between phases and heat capacities c_L, c_s as well as thermal conductivity k_L, k_s are constant.

$$\Delta x,\ \Delta t,\ \rho,\ c_L,\ c_s,\ k_L,\ k_s = const$$

Since we are dealing with a solid to liquid problem, the initial temperature profile of our mass will be assumed to be less than or equal to the melting temperature, as well as uniform for the sake of simplicity. If the energy reference temperature is taken as the melting temperature, then the initial energy is measured from a state where zero is the energy at melting point. Also, because the slab is initially a solid, the liquid fraction, $\lambda(x, 0) = 0$.

$$T_j^0 = T_{init} \leq T_m, \qquad j = 0, ..., M+1 \tag{7}$$

$$\left.\begin{aligned} E_j^0 &= \rho c_s (T_j^0 - T_m), \\ \lambda_j^0 &= 0, \end{aligned}\right\} \quad j = 1, ..., M \tag{8}$$

Conservation of energy is applied to each node independently to obtain a discrete heat balance. This balance is used to update the enthalpy, E_j, of each control volume; note that energy can only be conserved in a node and not on a boundary. The integral form

of the heat conduction equation can be transformed into a discrete and time explicit finite difference form. The statement is that the energy at t_{n+1} is equal to the energy at t_n plus the sum of the fluxes of the control volume at t_n.

$$E_j^{n+1} = E_j + \frac{\Delta t}{\Delta x}\left[q_{j-1/2}^n - q_{j+1/2}^n\right], \quad j = 1, ..., M \tag{9}$$

As shown above, the fluxes exist at the walls of each node so we must take care to account for information from each node that the wall belongs to. Internal fluxes exist at walls $j-1/2$ for $j = 2, ..., M$ and are represented with a discretized form of Fourier's law, where thermal resistance $R_{j\pm1/2}$ is a combination of the space and thermal conductivity along the conductive path. The resistance is to account for potentially different conductivity across the phase line.

$$q_{j-1/2} = -\frac{T_j - T_{j-1}}{R_{j-1/2}}, \quad R_{j-1/2} = \frac{\Delta x/2}{k_j} + \frac{\Delta x/2}{k_{j-1}}, \quad j = 2, ..., M \tag{10}$$

The outer boundary fluxes are denoted $q_{1/2}$ and $q_{M+1/2}$. Boundary conditions can be applied as constant temperatures, constant fluxes, or convective temperatures–each with their own relationships. The three cases for the left boundary represented below all have the same resistivity, and h is the convective heat transfer coefficient between the material and ambient gas. Similar relationships describe the end boundary conditions as well.

$$\left.\begin{aligned}
\textbf{Temperature:} \quad & T_0^n = T_0 = const, \quad q_{1/2}^n = -\frac{T_1^n - T_0}{R_{1/2}}, \\
\textbf{Flux:} \quad & q_{1/2}^n = q_{1/2} = const, \quad q_{1/2} = -\frac{T_1^n - T_0^n}{R_{1/2}}, \\
\textbf{Convective:} \quad & T_\infty^n = T_\infty = const, \quad q_{1/2}^n = -\frac{T_1^n - T_\infty}{\frac{1}{h} + R_{1/2}},
\end{aligned}\right\} R_{1/2} = \frac{1/2\Delta x}{k_1} \tag{11}$$

Once the temperatures are initialized, energies and phases are obtained, fluxes computed from temperatures and boundary conditions, one can march the solution forward by one step in time such that $t_{n+1} = t_n + \Delta t$. After updating the enthalpies within each volume, the temperatures and phases at each node can be extracted from the energy using the equation of state and the process can be repeated again. The energy required to change phase is defined as the latent heat, L; when a node overcomes this energy gap it is considered fully liquid so the method of temperature determination must change to agree with this.

$$T_j^n = \begin{cases}
T_m + \dfrac{E_j^n}{\rho c_s}, & E_j^n \le 0 \ \ \text{(solid)} \\
T_m, & 0 < E_j^n < \rho L \ \ \text{(interface)} \\
T_m + \dfrac{E_j^n - \rho L}{\rho c_L}, & E_j^n \ge \rho L \ \ \text{(liquid)}
\end{cases} \tag{12}$$

Liquid fraction can be conveniently determined by the same conditions as the temperatures above. Nodes will now be assigned the correct heat capacities and thermal conductivity. It is important to know when a node is mushy because it exists in a state of both liquid and solid, so the effective thermal conductivity must account for that.

$$\lambda_j^n = \begin{cases} 0, & E_j^n \leq 0 \quad \text{(solid)} \\ \dfrac{E_j^n}{\rho L}, & 0 < E_j^n < \rho L \quad \text{(mushy)} \\ 1, & E_j^n \geq \rho L \quad \text{(liquid)} \end{cases} \tag{13}$$

$$k_j^n = \left(\frac{\lambda_j^n}{k_L} + \frac{1 - \lambda_j^n}{k_s} \right)^{-1} \tag{14}$$

Moreover, the phase information is absolutely necessary in capturing the location of the phase front. We assume that for a mushy node, there exists a phase line somewhere within the node that delineates the boundary between solid and liquid. The phase front X^n is defined as the distance to the wall before the mushy node, where lowercase m denotes mushy, plus a factor of the liquid fraction of that mushy node times the spatial step. This definition requires there be one mushy node per time step thus the problem must reveal a sharp interface.

$$X^n := x_{m-1/2} + \lambda_m^n \Delta x \tag{15}$$

Before implementing the method in full, we must take a look at the stability of this scheme. Because the method is explicit in time, it is not unquestionably positivity preserving nor is it inherently stable for every choice of Δt. To guarantee stability, we must adhere to the CFL condition to guarantee no growth in errors. The stability condition follows from the pure heat conduction albeit a little more involved.

$$\begin{aligned} \Delta t_{max} &\leq \frac{1}{2} \frac{\Delta x}{\alpha_{max}^n} \\ \alpha_{max}^n &= max \left\{ \frac{k_L T_j^n}{\rho c_j}, \frac{k_s T_j^n}{\rho c_j}, \ j = 1, ..., M \right. \end{aligned} \tag{16}$$

As can be seen, the maximum time step is a function of the maximum temperature at some time t_n. This produces a tricky situation if one wishes to have a uniform time step all throughout, because one will have to predict a maximum temperature which will likely lead to some trial and error. Another issue is that this maximum temperature is not encountered

until later in the march, therefore extra and unnecessary calculations have had to be made until then due to the smaller time step. This can lead to unnecessary error accumulation due to round off.

To modify this one dimensional method into a two dimensional method, one simply must make a perpendicular mesh and be sure to account for two more fluxes into a given node. For a uniform mesh, the calculations are exactly the same.

The enthalpy method is great due its simplicity relative to other methods. It allows for capturing the phase boundary rather than tracking it which gets rid of the need for multiple coupled PDEs. There is a bit of an oversimplification, however; studies show that drag and surface tension effects dominate in a TIG welding process but our model ignores that. There is also a disregard for the change in density with temperature because materials in solid and liquid phases generally do not see a very big change over large temperature scales. That said, the final model gives a decent approximation for the weld pool characteristics for varying materials and power inputs.

4 Results

The code solves a 1-D heat transfer problem incorporating a phase change. For this example, the code solves a transient conduction problem in a undimensionalized fashion. The undimensional nature of the calculation allows for increased precision in the calculation due to the uneven distribution of numbers in computer systems. This system avoids error that could potentially arise from numbers near the limits of the chosen precision of the numbers stored in memory, i.e. single/double precision. In the figures shown, the temperature profiles are shown in undimensionalized form.

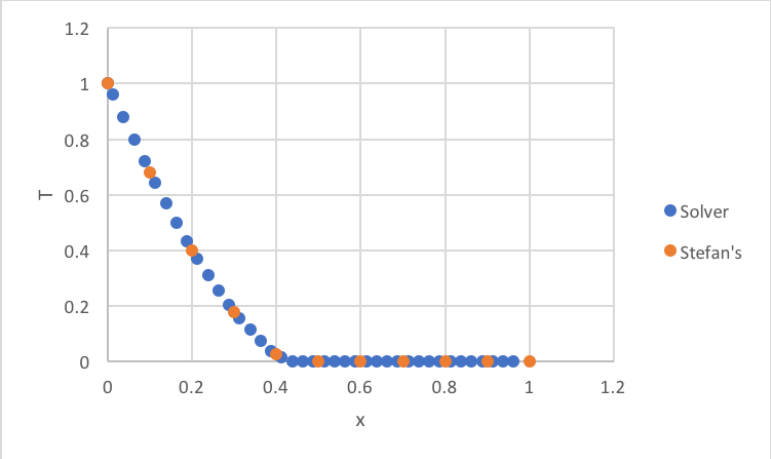


Figure 1: Solver vs. Stefan, t = 0.0402s

Actual results are shown below in Figures 1 and 2. Figure 1 shows both the explicit solver

solution and the Stefan Problem solution graphed on the same figures for the same inputs at the same time step. The initial conditions for each model is a hot singularity at the left end of the model with the rest of the nodes at a lower initial temperature. The thermal gradient is very high at the beginning of the time stepping as shown in Figure 1. Over time, however, the gradient evens out as shown in Figure 2. These results match what would happen in a real world conduction problem.

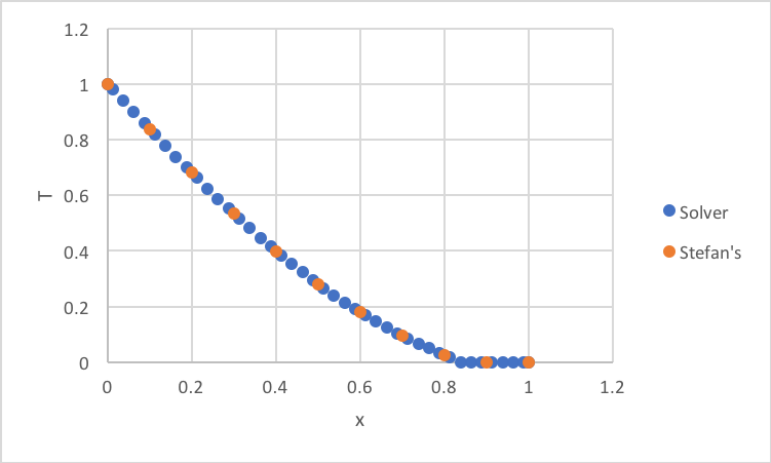


Figure 2: Solver vs. Stefan, t = 0.1600s

As a verification of the solver, the solution was compared to a solution derived from solving the Stefan Problem. For the two example time steps given, the solver exactly matches the solution provided by the Stefan Problem when provided with the same input parameters. This provides a measure of confirmation for the methods employed in modelling this conduction problem. The similarity of the solutions give enough confidence to pursue the explicit solver in two dimensions.

The input flux for all the cases follows a Gaussian distribution. The Gaussian is scaled by the power input, but otherwise remains the same distribution. A Gaussian function was chosen to approximate the distribution of the arc temperature in TIG welding. It matches the expected distribution relatively well for a stationary heat source.

All the simulations shared a set of assumed operating parameters. The initial and ambient parameters are outlined in Table 1. First, it was assumed that both the metal base and the air were at a room temperature value of 298K. A convection coefficient was assumed for all sides. These are held constant and shared for all of the cases. The materials studied are listed in Table 2. Because this study proposes to create a tool to analyze the behavior of non-familiar metals, the materials chosen are unique and non-existent (actually slightly modified from well-known metals at the author's whimsy). Alimium, ferisium, and tividium are the test materials, where alimium is the main focus of the numerical parameter study. The materials have unique differences and should prove to offer a varied range of behaviors.

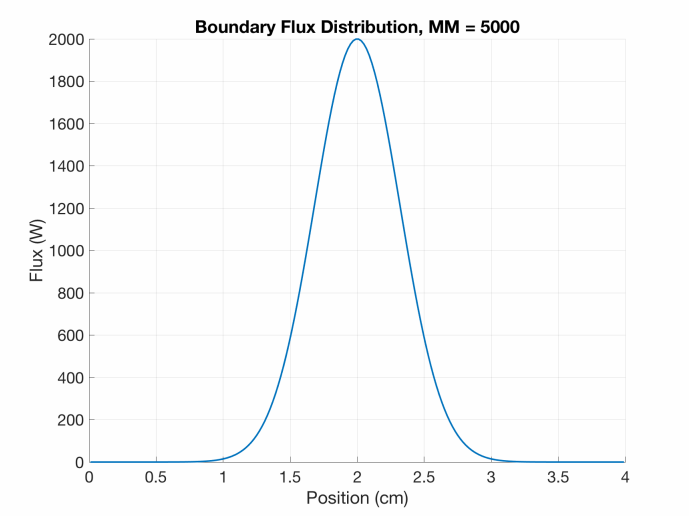


Figure 3: Gaussian Flux Distribution

Notably, alimium has a far greater thermal conductivity than either ferisium or tividium. The high thermal conductivity will mean diffusion will act at a much greater rate compared to the other materials. On the other hand, alimium also has the lowest melting temperature, with a value almost half that of ferisium or tividium. Tividium has vastly larger latent heat and specific heat, so it will likely be able to swallow heat from the torch and keep from melting as fast.

Shared Paramters				
T_0 (K)	T_{inf} (K)	h (W/m^2-K)	a (cm)	b (cm)
298	298	500	0	4

Table 1: Shared Parameters

Material	ρ (kg/m^3)	L (J/kg)	T_{melt} (K)	c_s (J/kg-K)	c_l (J/kg-K)	k_s (W/m-K)	k_l (W/m-K)
Alimium	2702	398	933	0.903	1.146	237	218
Ferisium	7870	272	1810	0.447	0.654	80.2	32.1
Tividium	4500	440000	1953	536	592	21	30

Table 2: Material Parameters

To begin, Table 3 shows the juxtaposed data of alimium, ferisium, and tividium at an MM of 5,000, and a torch power of 8,000 W. As predicted, tividium takes a lot longer to reach the desired weld pool size because of its rather large specific and latent heats. Because of this, there is more energy required to create a satisfactory pool then there is for the other two

metals. Tividium also does this relatively fast; when looking at the run time and comparing to the time for satisfactory pool size, the growth is not simply one to one. This is because the combination of material parameters results in a much larger time step, in fact being orders of magnitudes larger. The maximum temperature is also larger because the top face gets heated for so long. The minimum temperature at the end time hardly changes from the initial because the heat capacity is so large compared to the thermal conductivity. That allows the pool area to keep the enthalpy rather than pass it along, which leads to edges being neglected in terms of warmth. The maximum temperature for ferisium is relatively high compared to alimium as well. Unlike tividium, it has a low heat capacity compared to its thermal conductivity, but the melting point is similar to tividium and the maximum temperature is reflected in this as well. That coupled with a high material density makes for a low translation in temperature with respect to enthalpy, which leads to longer melt times and more heating. Finally, it is notable that all weld pools turn out to be the exact same depth regardless of material parameter; this is possibly due to the simplifications in our model, which imply a certain symmetrical independence from the physical parameters.

	Alimium	Ferisium	Tividium
MM	5000		
Power (W)	8000		
Energy (J)	40.08	135.88	49031.14
dt (s)	1.0295E-07	4.3864E-07	8.0400E-04
Iterations	48665	38552	7623
Time (ms)	5.01	16.91	6128.89
T_{max} (K)	1665.51	6562.15	7064.03
T_{min} (K)	469.36	570.48	298.07
Width (cm)	2.020	2.020	2.020
Depth (cm)	0.790	0.790	0.810
Run Time (s)	63.83	139.71	383.15

Table 3: Material Comparison

For the numerical parameter study (to test the uniqueness of solutions), we looked at two different scenarios for alimium: constant mesh density with varying flux and varying mesh density with constant flux. Relevant data for the constant mesh density, varying flux study may be found in Table 4 and illustrated for a few points in Figure 4. Data for the constant flux, varying mesh density study may be found in Table 5 and likewise for Figure 5. For both cases, the simulation was run until the weld pool reached a predefined size of 2 centimeters, roughly a typical value for a desired welding melt pool size.

For the constant Mesh size case, it was found that the time to run the simulation decreased as the power input increased. This effect results because less time is required to reach the desired weld pool size due to more power input. This means that fewer iterations are

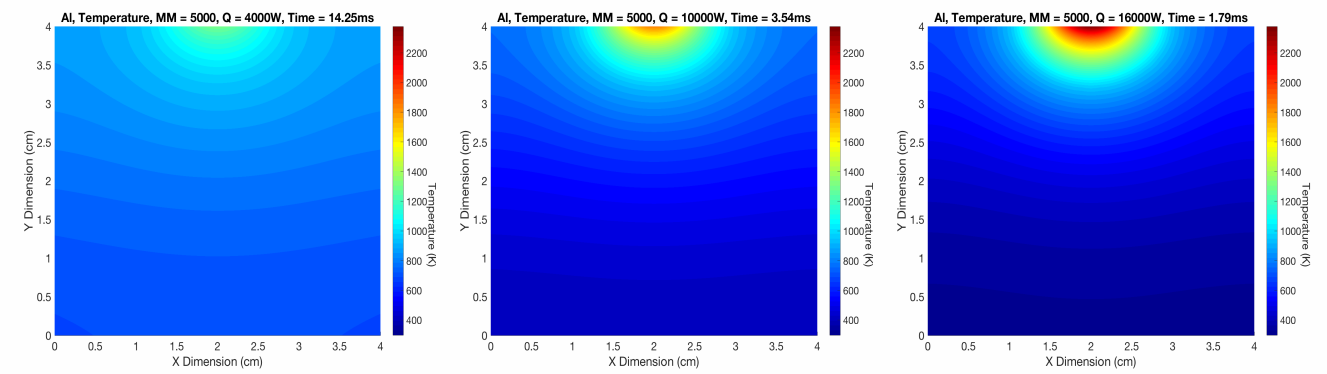
required which results in less simulation time and less run time. Since the mesh density is the same, then each iteration takes roughly the same amount of time to run, therefore fewer iterations result in a decreased run time. It should be noted that fewer iterations do not necessarily mean that the simulation is any less accurate; it simply means that the extra time is unnecessary. The discrepancy in time decreases with increasing power input, which implies an asymptotic approach. Moreover, higher torch flux results in decreasing total energy input. This ties in with the previous explanation. On the same table, notice how the maximum temperature keeps increasing with power, and that the minimum temperature keeps decreasing with power. The maximum temperature is obvious to explain; higher flux at the boundary leads to higher temperatures at the node adjacent to the boundary. The minimum temperature is a bit trickier, but falls in line with the previous discussion; temperature at the edges decrease because the time that the slab experiences heating is decreased. The heat simply does not have enough time to diffuse to the outer edges of the slab before the weld pool reaches the determined size. What is very interesting is that for the same weld pool width, the depth of the pool is also nearly exactly the same. The only different one being at 6,000 W and only by a few percent.

Alimium	MM = 5000, dt = 1.0295E-07 s							
Power (W)	2000	4000	6000	8000	10000	12000	14000	16000
Energy (J)	40.00	56.60	46.98	40.08	35.40	32.28	30.10	28.64
Iterations	194270	137446	76057	48665	34386	26130	20885	17388
Time (ms)	20.00	14.15	7.83	5.01	3.54	2.69	2.15	1.79
T_{max} (K)	915.76	1302.95	1484.66	1665.51	1845.75	2024.90	2201.70	2377.85
T_{min} (K)	590.72	684.37	567.46	469.36	401.96	360.12	334.87	320.16
Width (cm)	2.02	2.02	2.02	2.02	2.02	2.02	2.02	2.02
Depth (cm)	0.79	0.79	0.77	0.79	0.79	0.79	0.79	0.79
Run Time (s)	375.00	165.50	107.00	63.83	33.35	25.32	20.00	16.76

Table 4: Alimium; Constant MM

To further illustrate the point that lower torch power results in increased melt times, take a look at Figure 4. These graphs represent the temperature distribution in the slab when the weld pool has reached the desired valued of 2 cm in width. First, although the figures have different maximum and minimum temperatures, the temperature scale is the same for visual convenience. Notice that the edges of the 16 kW case (Fig. 4c) are a darker blue and therefore a lower temperature. This is different than for the 4 kW shown in Fig. 4a, where the edges are lighter. Even though the torch is at a lower setting, the block on the left ends up with a higher overall edge temperature. Again this is because the slab has enough time to diffuse the heat away from the lit up boundary. Figure 4 illustrates how this effect happens gradually with increasing flux.

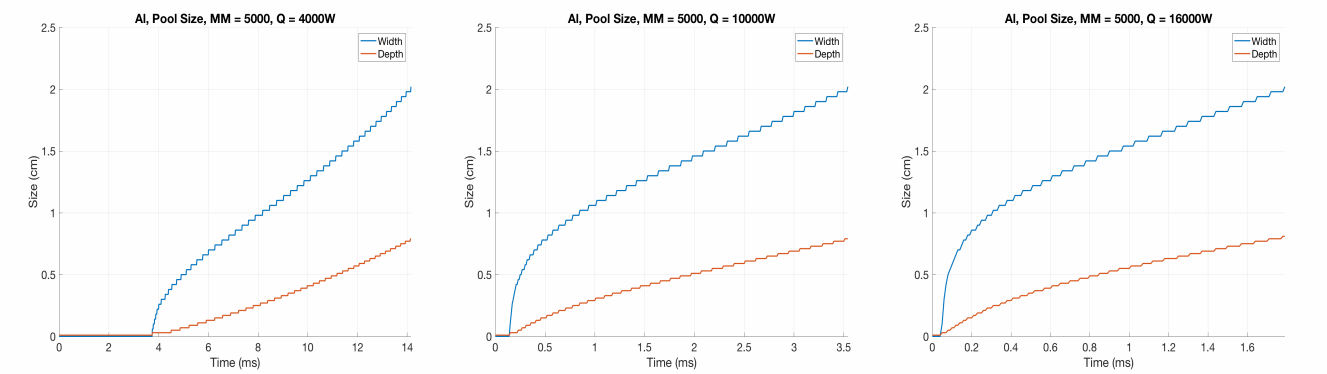
More evidence for this point is revealed in Figure 5, where a look at the time to induce melting is shown. For a best correlation, the test cases are exactly parallel to the ones in Figure 4. The 4 kW slab (5a) goes nearly 4 milliseconds before actually breaking the latent heat barrier. This is 4 milliseconds of pure diffusion and is rather significant when compared



(a) 4,000 W (b) 10,000 W (c) 16,000 W

Figure 4: Alimium; MM=5,000

to the other cases which have less than a quarter of a millisecond of diffusion. Thus, these plots coupled with the temperature contours speak for themselves it seems. In addition, the rate of growth seems to become much steeper initially, then dip off as power is increased. This could be due to lower bulk temperatures of the slab in the higher power cases. It seems that if the slab is preheated, the melting occurs more gradually at first then quicker when started, which is the opposite for a cold slab.



(a) 4,000 W (b) 10,000 W (c) 16,000 W

Figure 5: Alimium; Weld Pool Size; MM=5,000

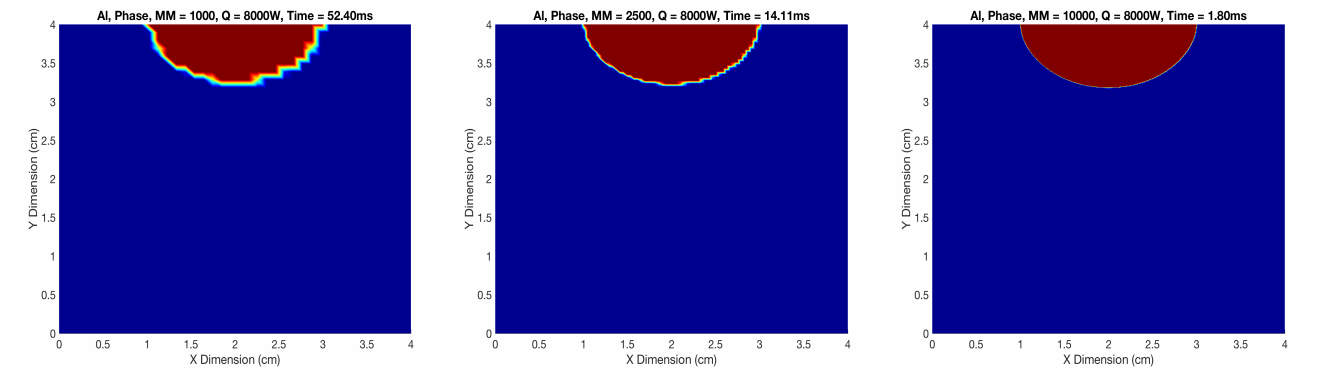
For the constant power case, it was found that the time needed for the weld pool to reach 2 cm decreased at a decreasing rate as the mesh was continuously refined. At the finest mesh tested, the weld pool reached 2 cm in 1.58 seconds, approaching asymptotically to a limit. It would be essential to determine when this change ceased to occur, that way a desired mesh size could be honed in on. Unfortunately, the code threw a "segmentation fault: 11" error when the mesh was refined further than MM=11,000, preventing further

experimentation with finer meshes. It is expected that the time to reach the desired weld pool sized would converge to a value near 1.3 milliseconds if the code would run with finer meshes. As a result of the ever shortening times, the energy input decreases as well, which makes perfect sense. It is observed that iterations increase with more refinement, which is reflected directly in the maximum possible time-step. As the CFL number decreases, the iterations rise in concert. This, of course, also results in higher run time. The maximum temperature increases with finer mesh, even at the lower exposure times–which explains why the melt times are so short. And as we have seen previously, higher maximum temperatures are coupled with lower minimum temperatures because of less diffusion. Just like the melt times, these discrepancies in temperature become smaller and smaller as mesh is refined. Unlike all of the previous observations, the weld pool depth seems to increase with finer mesh with decreasing disparity. Another asymptotic behavior that leads to the conclusion of a desired mesh size where further refinement does not yield any difference in accuracy.

Alimium	**Power = 8000 W**					
MM	1000	2500	5000	7500	10000	11000
Energy (J)	419.21	112.88	40.08	21.60	14.40	12.64
dt (s)	2.5737E-06	4.1179E-07	1.0295E-07	4.5755E-08	2.5737E-08	2.1271E-08
Iterations	20360	34265	48665	59010	69938	74281
Time (ms)	52.40	14.11	5.01	2.70	1.80	1.58
T_{max} (K)	1067.24	1293.00	1665.51	2035.60	2398.87	2539.52
T_{min} (K)	815.55	683.55	469.36	360.71	320.59	312.88
Width (cm)	2.100	2.040	2.020	2.013	2.010	2.009
Depth (cm)	0.780	0.750	0.790	0.807	0.815	0.823
Run Time (s)	13.27	22.14	63.83	87.83	156.57	196.00

Table 5: Alimium; Constant Power

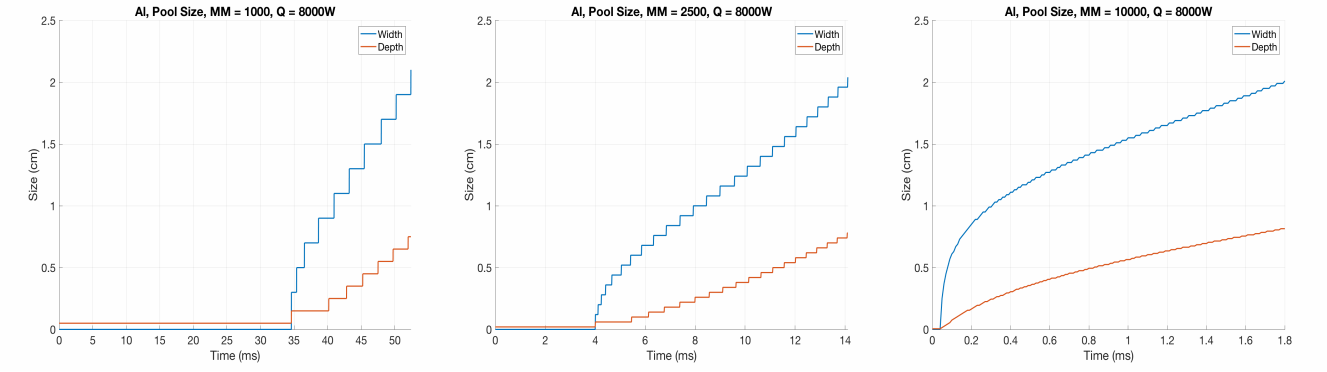
For a constant power input, it was found that an increasingly refined mesh provided a better defined phase transition. The phase front with the finer meshes (Figure 6b and 6c) are symmetrically curved, whereas the coarser meshes (Figure 6a) have a wavy appearance. Additionally, the mushy zone appears to be much larger and asymmetrically defined. This asymmetry could be the result of the Gaussian flux distribution being choppy and uneven at for coarse meshes. As the mesh is increasingly refined, the mushy zone becomes progressively smaller, eventually converging on a very sharp profile. It appears that the general width and depth remains the same. The mushy zone appears bigger simply because the mesh size is bigger, thus the nodes must take on a value that is likely a gross average of what would otherwise be a sharp front. The finer meshes shrink the observed phase lines closer to expected real world values, which points even more to the higher MM's leading to a desired mesh size. There will be a limit to the shrinking of the phase line, as there should not exist a discontinuity between liquid and solid. Note that a ten times refinement does a remarkable job at reducing the blob likeness from Figure 6a to 6c.



(a) MM = 1,000 (b) MM = 2,500 (c) MM = 10,000

Figure 6: Alimium; 8,000 W

Figure 7 compliments Figure 6 quite nicely. It, like Figure 5, shoes the growth of the weld pool's width and depth with respect to time. The coarse mesh size note only results in choppy size growth, it also keeps the material from melting for quite some time (as reflected in Table 5). It seems again that there is a behavior where later growth is steeper, but this time at lower mesh sizes. This is again due to longer diffusion times resulting in heating of the metal, which allows the pool to grow less inhibited. This could be a result of the Gaussian distribution being so choppy that the integrated heat is not the same for coarser meshes than it is for fine ones. It could also be that because the control volumes are bigger, they diffuse heat more rapidly, thus leading to the behavior shown. Either way, it is conclusive that finer meshes lead to higher accuracy results.



(a) MM = 1,000 (b) MM = 2,500 (c) MM = 10,000

Figure 7: Alimium; Weld Pool Size; 8,000W

5 Conclusion

The solutions compiled here are essentially a result of two separate cases of 2-D diffusion where parameters morph across an enthalpy barrier. The code runs as designed, and the ability to input arbitrary materials, mesh sizes, torch powers, temperature boundaries, the ability to choose between a Gaussian or uniform boundary condition, as well as automatic plot generation and animation capability, proves to show that the utilization and application of this tool is infinite in practice.

The main goal here was to show that the study of process parameters required for arbitrary materials can be hosted in a computational process, in lieu of the expensive and involved past-time of trial and error. There are certainly trends that must be observed, and from them rules that must be obeyed in order to achieve accurate results, which was the secondary goal of this report. We are confident in the results due to the unstrained manner in which we understood, logically, the behavior for different materials (absurd as they were to illustrate extremes).

Subsequent versions of this project, given a second round of funding, include many improvements. The prospect of super time-stepping for increased efficiency is a must for decreasing run times. The segmentation fault at high array sizes may be terminated by choosing alternate ways of passing to functions, which shall be explored. There are a few ways accuracy can be further increased. The inclusion of a vast variability in physical parameters with respect to temperature can help bring errors down. Finally, a closer look at other physical phenomena happening in the weld pool may offer some insight into how the model may benefit from such inclusions.

In order to provide an increased utility to the company and third party enthusiasts, the code is hosted in a public repository on GitHub at the URL: https://github.com/rekeshali/weldpool. The use of git for source control allows for revision history to be continuously saved and ensures that updates can be constantly implemented in local repositories. The code base is organized into descriptive directories to maintain order and organization. Additionally, a robust makefile is included to make compilation on Unix-like operating systems as easy as typing "make" on the command line. The makefile supports efficient compilation by compiling only what is necessary after changing individual files and then linking the newly compiled files with the other previously compiled files. The result is decreased compilation times and ease of use.

```
 //const int * W;
 int M, BCType, FType, nbar, maxwidth;
 double a, b;
 double dt, dx, dy;
 double Cs, Cl;
 double ks, kl;
 double rho, L;
 double h, Tinf;
 double T0, Q0, Tm;
#define pi 3.14159265358979323846
```

```
function animate(inputfile, outputfile, x, y, MM, Q, dtout, dtquit)
    gifname = sprintf('%s_%s_%d_%d.gif', inputfile, outputfile, MM,
Q); % name of gif
    command = sprintf('rm %s', gifname); % deletes old
    system(command); % does above
    [O, Omin, Omax, N] = getoutput(outputfile, 0); % gets min/max Temp
for scale

    h = figure;
    hold on
    set(gcf, 'Units', 'Normalized', 'OuterPosition', [0.1, 0.1, .6,
0.8]);
    set(gcf,'color','w');

    for i = 1:N
        surf(x,y,O(:,:,i));
        if(i == 1)
            if(outputfile(1:4) == 'temp') % no colorbar if phase
                c = colorbar;
                ylc = ylabel(c, 'Temperature (K)', 'FontSize', 20,
'Rotation', 270);
                posy = get(ylc, 'Position');
                set(ylc, 'Position', posy + [2, (Omax-Omin)/2 -
posy(2), 0]);
                caxis([Omin Omax]);
                ylabel('Y Dimension (cm)');
                xlabel('X Dimension (cm)');
            elseif(outputfile(1:5) == 'phase')
                ylabel('Y Dimension (cm)');
                xlabel('X Dimension (cm)');
            end
        end

        titstr = sprintf('time elapsed = %f s', (i-1)*dtout);
        if(outputfile(1:4) == 'temp') % no colorbar if phase
            titstr = sprintf('Temperature, %s', titstr);
            title(titstr);
        elseif(outputfile(1:5) == 'phase')
            titstr = sprintf('Phase, %s', titstr);
            title(titstr)
        end

        set(gca, 'FontSize', 20);
        colormap jet
        shading interp;
        clear figure
        view(0,90);
%         hold on;
        %pause(dtout/scale); % time accurate plot
```

```
        gifmaker(gifname, h, i); % saves gif
        if((i-1)*dtout > dtquit) % break at steady state
            break;
        end

    end
end
```

```
function data = fgetmat(fid)
% Rekesh Ali
% Function quits at eof or empty line!
i = 1;
while ~feof(fid)
    line = fgetl(fid);
    if isempty(line)
        break;
    end
    data(i,:) = sscanf(line, '%f');
        i = i + 1;
end
```

```
 filedir = sprintf("../outputs/flxdst.o"); % output file dir
    fid = fopen(filedir); % opens file
    picname = 'flxdst8000W1000MM.png';

    data = fgetmat(fid);
    fclose(fid);
    x = data(:,1);
    flux = data(:,2);

    fig = figure;
    hold on
    set(gcf, 'Units', 'Normalized', 'OuterPosition', [0.1, 0.1, .6,
0.8]);
    set(gcf,'color','w');
    plot(x*100, flux, 'LineWidth', 2);
%     xlim([0 max(time)]);
    xlabel('Position (cm)');
    ylabel('Flux (W)');
%     tit = sprintf('%s, Pool Size, MM = %.0f, Q = %.0fW', inputfile,
MM, Q);
    title('Boundary Flux Distribution, MM = 5000');
    set(gca, 'FontSize', 20);
    grid on

    frame = getframe(fig);
    im = frame2im(frame);
    [imind,cm] = rgb2ind(im,256);
    imwrite(imind,cm,picname,'png', 'WriteMode','overwrite');
```

```
function [O, Omin, Omax, i] = getoutput(outputfile, j)
    filedir = sprintf("../outputs/%s.o",outputfile); % output file dir
    fid = fopen(filedir); % opens file

    Omin = 1000000000000; % arbitrary initializers
    Omax = 0;
    i = 0;
    while ~feof(fid) % until the end of file
        i = i + 1;
        if j == 0 % animate
            O(:,:,i) = fgetmat(fid); % gets output matrix at every
time-step
            mintemp = min(min(O(:,:,i)));
            maxtemp = max(max(O(:,:,i)));
        else % just last one
            O = fgetmat(fid);
            mintemp = min(min(O));
            maxtemp = max(max(O));
        end


        if(maxtemp > Omax) % finds min and max Temps for plot scale
            Omax = maxtemp;
        end
        if(mintemp < Omin)
            Omin = mintemp;
        end
    end
    fclose(fid);
end
```

```
function gifmaker(gifname,figlabel,i)
    % Makes a gif, duh
    frame = getframe(figlabel);
    im = frame2im(frame);
    [imind,cm] = rgb2ind(im,256);
    % Write to the GIF File
    if i == 1
      imwrite(imind,cm,gifname,'gif', 'Loopcount',inf,
'WriteMode','overwrite');
    else
      imwrite(imind,cm,gifname,'gif','WriteMode','append');
    end
end
```

```
function plotend(inputfile, outputfile, x, y, MM, Q, tend)
    [O, Omin, Omax, ~] = getoutput(outputfile, 1);
    Omax
    Omin
    picname = sprintf('%s_%s_%d_%d.png', inputfile, outputfile, MM,
Q);
    %Omax = 7064.03;
    Omin = 298;
    fig = figure;
    hold on
    set(gcf, 'Units', 'Normalized', 'OuterPosition', [0.1, 0.1, .6,
0.8]);
    set(gcf,'color','w');

    surf(x,y,O);


    if(outputfile(1:4) == 'temp') % no colorbar if phase
        c = colorbar;
        ylc = ylabel(c, 'Temperature (K)', 'FontSize', 20, 'Rotation',
270);
        posy = get(ylc, 'Position');
        set(ylc, 'Position', posy + [2, (Omax-Omin)/2 - posy(2), 0]);
        caxis([Omin Omax]);
        ylabel('Y Dimension (cm)');
        xlabel('X Dimension (cm)');
    elseif(outputfile(1:5) == 'phase')
        ylabel('Y Dimension (cm)');
        xlabel('X Dimension (cm)');
    end

    if(outputfile(1:4) == 'temp') % no colorbar if phase
        titstr = sprintf('%s, Temperature, MM = %.0f, Q = %.0fW, Time
= %4.2fms', ...
            inputfile, MM , Q, 1000*tend);
        title(titstr);
    elseif(outputfile(1:5) == 'phase')
        titstr = sprintf('%s, Phase, MM = %.0f, Q = %.0fW, Time =
%4.2fms', ...
            inputfile, MM , Q, 1000*tend);
        title(titstr)
    end

    set(gca, 'FontSize', 20);
    colormap jet
    shading interp;
    clear figure
    view(0,90);
```

```
    frame = getframe(fig);
    im = frame2im(frame);
    [imind,cm] = rgb2ind(im,256);
    imwrite(imind,cm,picname,'png', 'WriteMode','overwrite');
end
```

```
function plotmain(inputfile,outputfile, dtquit, tend)
    % Example: plotmain('debug','temp') runs the temp output from the
debug
    % input file
    [x, y, dtout, ~, Tm, MM, Q] = readinput(inputfile); % gets mesh
and dtout
    plotpoolsize(inputfile, MM, Q);
    plotend(inputfile, outputfile, x, y, MM, Q, tend/1000);
%     animate(inputfile, outputfile, x, y, MM, Q, dtout, dtquit)
end
```

```
function plotpoolsize(inputfile, MM, Q)
    filedir = sprintf("../outputs/values.o"); % output file dir
    fid = fopen(filedir); % opens file
    picname = sprintf('%s_size_%d_%d.png', inputfile, MM, Q);

    fgetl(fid); % useless
    data = fgetmat(fid);
    fclose(fid);
    time = data(:,2);
    width = data(:,3);
    depth = data(:,4);
   % energy = data(:,5);

    fig = figure;
    hold on
    set(gcf, 'Units', 'Normalized', 'OuterPosition', [0.1, 0.1, .6,
0.8]);
    set(gcf,'color','w');
    plot(time,width, 'LineWidth', 2);
    plot(time,depth, 'LineWidth', 2);
    xlim([0 max(time)]);
    xlabel('Time (ms)');
    ylabel('Size (cm)');
    tit = sprintf('%s, Pool Size, MM = %.0f, Q = %.0fW', inputfile,
MM, Q);
    title(tit);
    legend({'Width','Depth'})
    set(gca, 'FontSize', 20);
    grid on

    frame = getframe(fig);
    im = frame2im(frame);
    [imind,cm] = rgb2ind(im,256);
    imwrite(imind,cm,picname,'png', 'WriteMode','overwrite');
end
```

```
function [x, y, dtout, tend, Tm, MM, Q] = readinput(inputfile)
    % Reads input file for automatic plot generation
    filename = sprintf('../inputs/%s.i', inputfile); % file directory
    fID = fopen(filename); % open file

    fgetl(fID); % useless data
    fgetl(fID);

    buff = fgetl(fID); % line with dtout
    data = strsplit(buff);
    dtout = str2double(data{2});% dtout secured
    tend = str2double(data{3}); % tend

    fgetl(fID); % more useless
    fgetl(fID);
    clear buff data
    buff = fgetl(fID); % line with MM, etc
    data = strsplit(buff);
    MM = str2double(data{1}); % dimensional data
    a = str2double(data{2});
    b = str2double(data{3});

    fgetl(fID); % more useless
    fgetl(fID);
    clear buff data
    buff = fgetl(fID); % line with Tm, etc
    data = strsplit(buff);
    Tm = str2double(data{3});

    fgetl(fID); % more useless
    fgetl(fID);
    fgetl(fID);
    fgetl(fID);
    clear buff data
    buff = fgetl(fID); % line with Tm, etc
    data = strsplit(buff);
    Q = str2double(data{2});


    M = MM*(b-a);
    gridline = linspace(a,b,M);
    [x,y] = meshgrid(gridline); % mesh for plotting secured
    x = x*100;
        y = y*100;
    fclose(fID); % being neat
end
```

```
#include "global.h"
#include <math.h>
#include <stdio.h>
void BCFlux(double F0[]){
        // creates a flux vector that holds value at each node
        // corresponding to gaussian distribution
        FILE * OUT;
        OUT = fopen("outputs/flxdst.o", "w"); // output flux
distribution

        double x;
        double center = 0.5*(b-a); // location of peak
        double scale = 5/(b); // will look like gaussian of 5 max
        double stddev = 1/(scale*sqrt(2*pi)); // width of flux
        for(int i = 1; i <= M; i++){ // for every node
                x = a + ((double)i - 0.5)*dx; // location of current
node
                if(!FType){ // Gaussian distribution
                        F0[i] = Q0*exp( -(x - center)*(x - center)/
(2*stddev*stddev)); // flux at node
                }else{ // Uniform distribution
                        F0[i] = Q0; // all flxues same
                }
                fprintf(OUT, "%.6f %11.4f\n", x, F0[i]); // keeping
track
        }
        fclose(OUT);
}
```

```
//================================== Compute conductivities
#include "global.h"
double conduct(double p){
    // check liquid fraction of node
    if(p == 0){
        return ks;
    }else if(p == 1){
        return kl;
    }else{
        double kinv =  p/kl + (1-p)/ks; // mushy
        return 1/kinv;
    }
}
```

```
// declare functions to get rid of compiler warnings
void readfile(char filename[], double *factor, double *dtout, double
*tend, int *MM);
void mesh(double X[], double Y[]);
void init(const int W, double T[][W+2], double E[][W+1], double p[]
[W+1]);
void BCFlux(double F0[]);
void output(const int W, double X[], double Y[], double T[][W+2],
double Fx[][W+1], double Fy[][W+1],
            double E[][W+1], double p[][W+1], double time, int nsteps,
double tend);
void flux(const int W, double Fx[][W+1], double Fy[][W+1], double T[]
[W+2], double p[][W+1], double F0[]);
void pde(const int W, double E[][W+1], double Fx[][W+1], double Fy[]
[W+1]);
void eos(const int W, double E[][W+1], double T[][W+2], double p[]
[W+1], double Fx[][W+1], double Fy[][W+1]);
```

```
//============================== EQUATION OF STATE
#include "global.h"
#include <stdio.h>

double conduct(double p);  // declare function

void eos(const int W, double E[][W+1], double T[][W+2], double p[]
[W+1], double Fx[][W+1], double Fy[][W+1]){
        // Nodes
        for(int i = 1; i <= M; i++){
                for(int j = 1; j <= M; j++){
                        if(E[i][j] < 0){
                                //if (i==i){printf("im solid\n");}
                                T[i][j] = Tm + E[i][j]/(rho*Cs);
                                p[i][j] = 0;
                        }else if(E[i][j] >= 0 && E[i][j] <= rho*L){
                                //printf("im mushy\n");
                                T[i][j] = Tm;
                                p[i][j] = E[i][j]/(rho*L);
                        }else if(E[i][j] > rho*L){
                                //printf("im liquid\n");
                                T[i][j] = Tm + (E[i][j] - rho*L)/
(rho*Cl);
                                p[i][j] = 1;
                                //printf("%i frac=%f",i, p[i]);
                        }
                /*if (i ==i){
                        printf("Tn+1=%f en+1=%f\n\n", T[1], E[0]);
                }*/
                }
        }

        /*printf("enafterloop=%f\n", E[0]);
        E[0] = E0;
        if(p[1] == 1){p[0] = 1;}*/
        // Boundaries
        double k, R;
                // LEFT AND RIGHT
        for(int j = 1; j <= M; j++){
                k = conduct(p[1][j]);
                R = dx/(2*k);
                T[0][j] = Fx[0][j]*R + T[1][j];
                        // RIGHT
                k = conduct(p[M][j]);
                R = dx/(2*k);
                //T[M+1] = (R*h*Tinf - T[M])/(R*h - 1);
                T[M+1][j] = T[M][j] - Fx[M][j]*R;
                //T[M+1][j] = T0;
        }
```

```
                // DOWN AND UP
        for(int i = 1; i <= M; i++){
                k = conduct(p[i][1]);
                R = dx/(2*k);
                T[i][0] = Fy[i][0]*R + T[i][1];
                        // RIGHT
                k = conduct(p[i][M]);
                R = dx/(2*k);
                //T[M+1] = (R*h*Tinf - T[M])/(R*h - 1);
                T[i][M+1] = T[i][M] - Fy[i][M]*R;
                //T[i][M+1] = T0;
        }

// CORNERS
T[0][0] = (T[0][1] + T[1][0])/2.;
T[0][M+1] = (T[0][M] + T[1][M+1])/2.;
T[M+1][M+1] = (T[M][M+1] + T[M+1][M])/2.;
T[M+1][0] = (T[M][0] + T[M+1][1])/2.;

}
```

```
//========================================== create fluxes at faces
between CV's
#include "global.h"
#include <math.h>
#include <stdio.h>

double conduct(double p);  // declare function

void flux(const int W, double Fx[][W+1], double Fy[][W+1], double T[]
[W+2], double p[][W+1], double F0[]){
        double k, ki, kim, R;
        // Boundaries
                // Left and right
        for(int j = 1; j <= M; j++){
                k = conduct(p[1][j]);
                R = dx/(2*k);
                //Fx[0][j] = -(T[1][j] - Q0)/R; //F_1/2
       Fx[0][j] = - (T[0][j] - Tinf)/(R + 1/h); // left; convective
2-D
                k = conduct(p[M][j]);
                R = dx/(2*k);
                //F[M] = - (T[M] - Tinf)/(R + 1/h); // convective 1-D
                //Fx[M][j] = - (T[M+1][j] - T[M][j])/R; // insulated
2-D
                Fx[M][j] = (T[M][j] - Tinf)/(R + 1/h); // right;
convective 2-D
        }
                // Bottom and top
        for(int i = 1; i <= M; i++){
                k = conduct(p[i][1]);
                R = dx/(2*k);
                //Fy[i][0] = -(T[i][1] - Q0)/R; //F_1/2
       Fy[i][0] = - (T[i][0] - Tinf)/(R + 1/h); // bottom; convective
2-D
                //==================== TOP
=======================================
                k = conduct(p[i][M]);
                R = dx/(2*k);
                if(!BCType){ // checks boundary condition type
                        Fy[i][M] = (T[i][M] - F0[i])/R; // const temp
BC
                        //printf("TEMPBC, Flux = %f\n", Fy[i][M]);
                }else{
                        Fy[i][M] = - F0[i]/(dx);  // const Flux BC
                        //printf("FLUXBC, Flux = %f\n", Fy[i][M]);
                }
                //Fy[i][M] = (T[i][M] - Tinf)/(R + 1/h); //
convective 2-D
        }
```

```
        // LEFT TO RIGHT FLUX
        for(int i = 2; i <= M; i++){ // 2-1/2 < i-1/2 < M-1/2
                for(int j = 1; j <= M; j++){
                        // get k and R
                        ki = conduct(p[i][j]); // ki
                        kim = conduct(p[i-1][j]); // ki-1
                        R = dx/(2*ki) + dx/(2*kim); // Ri-1/2
                        // compute flux
                        Fx[i-1][j] = -(T[i][j] - T[i-1][j])/R; //
F_i-1/2
                }
        }
        // UP TO DOWN FLUX
        for(int i = 1; i <= M; i++){ // 2-1/2 < i-1/2 < M-1/2
                for(int j = 2; j <= M; j++){
                        // get k and R
                        ki = conduct(p[i][j]); // ki
                        kim = conduct(p[i][j-1]); // ki-1
                        R = dx/(2*ki) + dx/(2*kim); // Ri-1/2
                        // compute flux
                        Fy[i][j-1] = -(T[i][j] - T[i][j-1])/R; //
F_i-1/2
                }
        }
}
```

```
//================================== CREATE INITIAL PROFILE
#include "global.h"
#include <stdio.h>
void init(const int W, double T[W+1][W+2], double E[W+1][W+1], double
p[W+1][W+1]){
        // for nodes and boundaries
        for(int i = 0; i <= M+1; i++){
                for(int j = 0; j<= M+1; j++){
                        // Temperatures
                        T[i][j] = T0; // IC's
                        // Energies and liquid fractions
                        if(i > 0 && j > 0 && i < M+1 && j < M+1){ //
NODES ONLY
                                E[i][j] = rho*Cs*(T[i][j] - Tm);
                                p[i][j] = 0;
                        }
                }
        }
}
```

```
// REKESH ALI
// M475 TEAM F
// Conservation PDE

#include <stdio.h>
#include <math.h>
#include "global.h" // includes all subroutines
#include "declarations.h"  // to get rid of compiler warnings

int main(int argc, char * argv[]){
        //============================ Initialize I/O
        char inp[20];
        if(argc == 1){
                sprintf(inp, "input");
        }else{
                sprintf(inp, "%s.i", argv[1]);
        }

        //============================ READ INPUTS
        int MM;
        double tend, dtout, factor;
        readfile(inp, &factor, &dtout, &tend, &MM); // reads from
filename inp
        //============================ CREATE MESH
        M = (double)MM*(b-a); // number of CV's
        M = (int)M;
        const int W = M;
        dx = 1./((double)(MM)), dy = 1./((double)(MM)); // spacing
between nodes
        double X[M+2], Y[M+2]; // nodal array
        mesh(X, Y); // fills array with positions of nodes

        //============================ SET TIMESTEP
        double t0 = 0.0; // start time
        double kmax = fmax(kl, ks);
        double Cmin = fmin(Cl, Cs);
        double dtEXPL = dx*dx*rho*Cmin/(4.*kmax);
        dt = factor*dtEXPL; // dt fraction of CFL for stability
purposes
        int Nend = (int)((tend - t0)/dt) + 1; // number of timesteps
        double Nend2 = ((tend - t0)/dt) + 1;
        int nsteps = 0; // initialize timestep
        double time = t0; // initialize time
        double tout = fmax(dtout, dt); // time for printing to fil

        printf("M = %i, tend = %f, dt = %.15e, Nend = %i\n", M, tend,
dt, Nend);

        //============================ INITIALIZE PROFILE
        double T[M+2][M+2], E[M+1][M+1], p[M+1][M+1]; // solution
```

```
array and max error
        init(W, T, E, p); // fills solution array
        //=========================== BEGIN TIMESTEPPING
        double Fx[M+1][M+1], Fy[M+1][M+1]; // initialize flux array
        double F0[M+1]; // flux distribution array at boundary
        BCFlux(F0); // gaussian vs uniform distribution
        nbar = 1;
        maxwidth = 0;
        output(W, X, Y, T, Fx, Fy, E, p, time, nsteps, tend); // print
to file

 //   #pragma omp parallel for num_threads(4) schedule(dynamic)  //
parallel for loop
        for(nsteps = 1; nsteps <= Nend; nsteps++){
                time = nsteps*dt; // current time
                flux(W, Fx, Fy, T, p, F0); // current flux at walls
                pde(W, E, Fx, Fy); // updates energy with forward
euler
                eos(W, E, T, p, Fx, Fy); // updates temperatures and
phases
                if(time > tout){ // when time to print
                        output(W, X, Y, T, Fx, Fy, E, p, time,
nsteps, tend); // print to file
                        tout = tout + dtout; // next print time
        }
                if(maxwidth){
                break;}
        }
}
```

```
// create 2D mesh array
#include "global.h"
void mesh(double X[], double Y[]){
        X[0] = a; // boundaries
        X[M+1] = b;
        Y[0] = a;
        Y[M+1] = b;
        for(int i = 1; i <= M; i++){
                X[i] = a + ((double)i - 0.5)*dx; // within space
                Y[i] = a + ((double)i - 0.5)*dy; // within space
        }
}
```

```
//======================== print solution profile to text file at
certain times
#include "global.h"
#include <stdio.h>
void output(const int W, double X[], double Y[], double T[][W+2],
double Fx[][W+1], double Fy[][W+1], double E[][W+1], double p[][W+1],
double time, int nsteps, double tend){
//      double flux, energy, phase;
        FILE *OUT;
        FILE *TEMP;
        FILE *PHASE;
        FILE *ENTH; // initialize file var
        if(!nsteps){ // if haven't begun time stepping
                OUT = fopen("outputs/values.o", "w"); // new file
                fprintf(OUT,"#nstep time (ms)  width (cm) depth(cm)
energy (J)\n");
                TEMP = fopen("outputs/temp.o", "w"); // new file
                //fprintf(TEMP,"#Temperatures by timesteps\n");
                PHASE = fopen("outputs/phase.o", "w"); // new file
                //fprintf(PHASE,"#Liquid fraction by timestep\n");
                ENTH = fopen("outputs/enth.o", "w"); // new file
                //fprintf(ENTH,"#Enthalpies by timestep\n");
        }
        else{
                OUT = fopen("outputs/values.o", "a"); // old file
                TEMP = fopen("outputs/temp.o", "w"); // old file
                PHASE = fopen("outputs/phase.o", "w"); // old file
                ENTH = fopen("outputs/enth.o", "w"); // old file
        }
        // hidden time, steps, error
        // position and profile in columns
        //================================================
TEMPERATURES
        // inside temperatures only, walls not necessary
        for(int j = 1; j <= M; j++){
                for(int i = 1; i <= M; i++){
                        fprintf(TEMP, "%22.15e ", T[i][j]);
                }
                fprintf(TEMP, "\n");
        }
        fprintf(TEMP, "\n");
        fclose(TEMP);
        //================================================ PHASES
                for(int j = 1; j <= M; j++){
                        for(int i = 1; i <= M; i++){
                                fprintf(PHASE, "%22.15e ", p[i][j]);
                        }
                fprintf(PHASE, "\n");
        }
        fprintf(PHASE, "\n");
```

```
        fclose(PHASE);
        //================================================ ENTHALPIES
        // just for funsies
        for(int j = 1; j <= M; j++){
                for(int i = 1; i <= M; i++){
                        fprintf(ENTH, "%22.15e ", E[i][j]);
                }
                fprintf(ENTH, "\n");
        }
        fprintf(ENTH, "\n");
        fclose(ENTH);
        //================================================
TIMESTAMPS/ETC
        double xl, xr, yd, width, depth, energy;
        int il = M/2, ir = M/2, id = M;
        for(int l = M/2; l <= M; l++){
                for(int k = 1; k <= M; k++){ // bounds for width of
pool
                        if(p[k-1][l] < 1 && p[k+1][l] == 1){
                                if(k < il){
                                        il = k;}
                        }
                        if(p[k-1][l] == 1 && p[k+1][l] < 1){
                                if(k > ir){
                                        ir = k;}
                        }
        }
        }
        for(int k = 1; k <= M; k++){
                if(p[M/2][k-1] < 1 && p[M/2][k+1] == 1){
                        id = k;}
        }
        xl = a + ((double)il - 0.5)*dx;
        xr = a + ((double)ir - 0.5)*dx;
        yd = a + ((double)id - 0.5)*dx;
        width = 100*(xr - xl); // in cm
        depth = 100*(b - yd); // in cm
        energy = Q0*time; // in Joules
        fprintf(OUT, "%i %6.4f %6.4f %6.4f %6.4f\n", nsteps,
1000*time, width, depth, energy);
        //fprintf(OUT, "# Error up to this time: %.15e\n\n", ERR);
        //
        // COMPLETION BAR
        if(nsteps == 0){
                printf("0%%       20%%       40%%       60%%
80%%        100%%\n");
                printf(" ");
        }
        if( time > nbar*tend/50 ){
                //printf("%3.0f%%\n", 100*time/tend);
```

```
                printf("=");
                fflush(stdout);
                nbar = nbar + 1;
        }
        if(width > 2){
                maxwidth = 1;
                int barleft = 50 - nbar;
                for(int g = 0; g <= barleft; g++){
                printf("=");}
        }


        /*for(int j = M+1; j >= 0; j--){
                for(int i = 0; i <= M+1; i++){
                        fprintf(OUT, "%22.15e ", T[i][j]);
                }
                fprintf(OUT, "\n");
        }*/


        /*for(int i = 0; i <= M+1; i++){
                if(i == 0){
                        fprintf(OUT, "%18.15e %18.15e %18.15e\n",
X[i], T[i], F[i]);
                }
                if(i > 0 && i <= M){
                        fprintf(OUT, "%18.15e %18.15e %18.15e %18.15e
%.3f\n", X[i], T[i], F[i], E[i], p[i]);
                }else if(i > M){
                        fprintf(OUT, "%.15e %.15e\n", X[i], T[i]);
                }
        }*/
        //fprintf(OUT, "\n"); // new line to separate times of
printing
        fclose(OUT);
}
```

```
//========================================= Forward Euler (explicit)
#include "global.h"
void pde(const int W, double E[][W+1], double Fx[][W+1], double Fy[]
[W+1]){
    // Energy is conserved within nodes only..
        for(int i = 1; i <= M; i++){
                for(int j = 1; j <= M; j++){
                        E[i][j] = E[i][j] + (dt/dx)*(Fx[i-1][j] -
Fx[i][j]) + (dt/dy)*(Fy[i][j-1] - Fy[i][j]);
                        // Ei + dt/dx(sum fluxes)
                }
        }
}
```

```
//========================== READ INPUTS
#include "global.h"
#include <stdio.h>
void readfile(char filename[], double *factor, double *dtout, double
*tend, int *MM){
        FILE *IN;
    int n = 255;
    char buff[n];
    IN = fopen(filename, "r");

    // TIME
    fgets(buff, n, IN); // section
    fgets(buff, n, IN); // labels
    fscanf(IN, "%lf %lf %lf\n", factor, dtout, tend); // values

    // SPACE
    fgets(buff, n, IN);
    fgets(buff, n, IN);
    fscanf(IN, "%i %lf %lf\n", MM, &a, &b);

    // MATERIAL
    fgets(buff, n, IN);
    fgets(buff, n, IN);
    fscanf(IN, "%lf %lf %lf %lf %lf\n", &rho, &L, &Tm, &h, &Tinf);
    fgets(buff, n, IN);
    fscanf(IN, "%lf %lf %lf %lf\n", &Cs, &Cl, &ks, &kl);

    // IBCs
    fgets(buff, n, IN);
    fgets(buff, n, IN);
    fscanf(IN, "%lf %lf %i %i\n", &T0, &Q0, &BCType, &FType);

    // printf("%i, %f, %f, %f, %f\n", MM, Cs, b, Q0, T0);
    fclose(IN);
}
```