Table of Contents

1. Introduction and Description of the Code

Smooth Particle Fluid with MAXwell equation solver (SPFMax) is a three dimensional fluid code designed to model problems in inertial fusion, pulsed power, and propulsion. SPFMax numerically integrates three components of position and momentum and either single temperature or two temperature energy equations on a distributed set of point masses in which the derivatives are evaluated using smooth particle hydrodynamic methods. The physics include real viscosity, tabular or ideal gas equation of state, heat transfer, shock capturing via artificial viscosity, thermal equilibration between ions and electrons, and single and multigroup radiation emission and absorption. Linear momentum and all particle-to-particle energy interactions conserve momentum and energy exactly by enforcing reciprocity between particle pairs. This is accomplished by averaging interactions between particles. Electromagnetic fields are propagated by a network of transmission line equations based on Kirchoff's voltage and current laws, with vacuum field propagation handled by efficient superposition of charge and current density. SPFMax has been written in Matlab, and supports both multicore and GPU computing. All post processing was done with a custom GUI-based tool called SPHplot also developed in Matlab. Both SPFMax and SPHplot can produce synthetic diagnostics such as synthetic line integrated interferometry.

The objective of this document is to provide step by step and narrative details regarding use of and debugging of common problems in SPFMax. SPFMax is a meshless, Lagrangian, 3D hydrocode with Maxwell equation solver and self-consistent circuit model.

1.1 Smooth Particle Hydrodynamic Method (SPH): Integral Approximation

At the very core of this code is the smooth particle hydrodynamic (SPH) method, in which properties are approximated with the integral or kernel approximation according to

$$A_a(r)=\int A(r')W(r-r',h)\,dr' \tag{1}$$

where A is any property (like temperature, pressure, etc), subscript a means point a, r is the position of point a in space, and W is the interpolating kernel function. In the limit that $h\rightarrow 0$, W becomes the direct delta function and the expression becomes exact.

1.2 SPH Summation Approximation

Any function can be approximated in this way, and this is the first assumption of SPH. The second is to replace the integral with a summation,

$$A_a=\sum_b A_b V_b W_{ab}(r-r',h) \tag{2}$$

where V_b is the volume of the neighboring particles b. This is called the summation or particle approximation. This is done because we can't solve the problem at every point in space because computers are finite. Boo! The kernel function W is usually a Gaussian-like or cubic b-spline function which goes to zero at some κh, where $\kappa=2$ normally.

1.3 SPH Derivatives

Gradients can be approximated as

$$\nabla A_a = \sum_b A_b V_b \nabla W_{ab}(r - r', h) \tag{3}$$

1.4 SPH Kernel Functions

In SPFMax, the cubic spline function is used,

$$W_{ab} = \begin{cases} \frac{1}{4\pi h_{ab}^3}\left[(2-q)^3 - 4(1-q)^3\right], & \text{for} \quad 0 \le q \le 1 \\ \frac{1}{4\pi h_{ab}^3}(2-q)^3, & \text{for} \quad 1 \le q \le 2 \\ 0 & \text{for} \quad q > 2 \end{cases} \tag{4}$$

1.5 SPH: Finding the compact support distance (h) for each particle

The key to implementing any SPH method properly is to have an accurate list of neighbors for each particle and a compact support distance h which scales the kernel function and its gradients so the following constraints are satisfied

$$\sum_b \mathcal{V}_b W_{ab} = 1 \tag{5}$$

and

$$\sum_b \mathcal{V}_b \nabla W_{ab} = 0 \tag{6}$$

The particle volume is

$$\mathcal{V}_a = \frac{h_a^3}{\eta^3} \tag{7}$$

and η is 1.11. **It cannot be emphasized enough that if Eqs 5 and 6 are not satisfied, the code is not going to produce accurate results at all.** In practice, it is very difficult to choose a compact support distance for each particle to satisfy this everywhere. In SPFMax, h is uniquely found for each particle to satisfy the first constraint, Eq 5, to within 1%, and then the weights W_{ab} and ∇W_{ab} are scaled so that Eqs 5 and 6 are satisfied to within machine accuracy.

For clarity, this procedure is described below. First, particle position is tracked with

$$\frac{\partial \mathbf{r}}{\partial t} = \mathbf{u} \tag{8}$$

Next, the particle neighbors have to be determined, and SPFMax uses the k-d tree to return the 60 nearest neighbors for each point in a script called `update_neighbors.m`. The commands below are used

```
p.NS = createns(pts);
[p.nbrs,p.r] = knnsearch(p.NS,pts,'K',60);
```

where 'pts' stores the x,y,z information, and the nbrs and Euclidean distances are stored in 'nbrs' and 'r', respectively. To facilitate vector calculations, each particle has 60 neighbors so the nbrs matrix is a uniform, fixed block of memory. This is done even though the outer 15 or so rarely contribute to the summation interpolant, for computational speedup at the expense of memory.

Once this has been done, h first is estimated to be half of the maximum distance between the particle and its neighbors. An iterative Newton-Raphson method is then used to improve the h estimate as follows. The particle volume is evaluated at h+δh and h-δh, in which δh is ~h/.01. After this is done, the cubic spline kernel function is evaluated for h+δh and h-δh. Equation 5 is then used to evaluate the particle consistency at the two values for h, and this is used to find the difference between the two summations. The value for h is then updated with

$$h_a = \frac{2\delta h_a}{\left(\sum_b \mathcal{V} W_{ab}\right)_{h_a+\delta h} - \left(\sum_b \mathcal{V} W_{ab}\right)_{h_a-\delta h}} \operatorname{sign}\left[1 - \sum_b V_b W_{ab}\right] \min\left(\left|1 - \sum_b V_b W_{ab}\right|, 0.25\right) \tag{9}$$

This entire process is then repeated five times. The kernel weights (W_{ab}) are then scaled uniformly for the neighbors to force particle consistency, and this scaling change the value of h by up to~1%. Once h is determined, the particle masses on the first time step should be determined to be consistent with the initial density specified in the input file, and calculation of density is described below. The linear consistency constraint can also be enforced exactly by scaling either all of the negative or positive contributions to the summation of $\mathcal{V}_b \nabla W_{ab}$ for each particle over its neighbors. While numerous authors have explored more mathematically rigorous ways of achieving this, we found this to be very robust for our code.

2. Equations of motion

2.1 Conservation of Mass

SPFMax solves the single fluid equations of motion. Conservation of mass is traditionally given by

$$\frac{d\rho}{dt} + \rho \nabla \cdot (\mathbf{u}) = 0 \tag{10}$$

where ρ (kg/m^3) is the mass density, **u** is the velocity vector, and t is time and d represents the material derivative. SPFMax solves conservation of mass exactly density is determined by the particle mass divided by the particle volume, where the mass is a constant property of the particles,

$$\rho_a = \sum_b m_b W_b \tag{11}$$

2.2 Particle Position

Particle positions are updated with

$$\frac{d\mathbf{r}}{dt} = \mathbf{u} \tag{12}$$

2.3 Conservation of Linear Momentum

The momentum equation for a single fluid is given by

$$\frac{d\mathbf{u}}{dt} = -\frac{1}{\rho}\nabla p + \frac{1}{\rho}\nabla\cdot\boldsymbol{\tau} + \frac{1}{\rho}\mathbf{j}\times\mathbf{B} \tag{13}$$

where p is the static pressure and τ is the deviatoric viscous stress tensor.

2.4 Conservation of internal energy

2.4.1 Single Temperature Model

The single temperature energy equation ($T_i = T_e = T$) is given by

$$\frac{de}{dt} = -\frac{p}{\rho}\nabla\cdot\mathbf{u} + \frac{\boldsymbol{\tau}}{\rho}\nabla\cdot\mathbf{u} + \frac{1}{\rho}\nabla\cdot(k\nabla T) - 4\sigma T^4\chi_{Planck} + \frac{\eta}{\rho}j^2 \tag{14}$$

where k is the thermal conductivity, σ is the Stefan-Boltzmann constant, T is temperature, and χ_{Planck} is the single group Planck emission opacity.

Alternatively, if the optical thickness $\frac{1}{\rho\chi_{Planck}}$ is of the same order or smaller than the particle scale h, then radiation can be modeled as a diffusion process by adding an additional term to the overall thermal conductivity

$$k_{total} = k + k_{Ross} = k + \frac{4acT_e^3}{3\rho\chi_{Ross}} \tag{15}$$

where $a = 4\sigma/c$ is the radiation density constant.

2.4.2 Two Temperature Model

If the temperature is split between ions and electrons, then the energy equations are given by

$$\frac{\partial e_i}{\partial t} = -\frac{p_i}{\rho}\nabla\cdot\mathbf{u} + \frac{\boldsymbol{\tau}}{\rho}\nabla\cdot\mathbf{u} + \frac{1}{\rho}\nabla\cdot(k_i\nabla T_i) + Q_{ei} \tag{16}$$

$$\frac{\partial e_e}{\partial t} = -\frac{p_e}{\rho}\nabla\cdot\mathbf{u} + \frac{1}{\rho}\nabla\cdot(k_e\nabla T) - 4\sigma T_e^4\chi_{Planck} - Q_{ei} + \frac{\eta}{\rho}j^2 \tag{17}$$

where

$$Q_{ei} = \left(\frac{\partial e}{\partial t}\right)_{ion} = -\left(\frac{\partial e}{\partial t}\right)_{el} = \frac{3}{2}\frac{k}{m_i}\frac{(T_e - T_i)}{\tau_{ei}} \tag{18}$$

and the electron collision time is

$$\tau_{ei} = 1.40527 \times 10^{11} \frac{\left(m_e T_i + m_i T_e\right)^{3/2}}{\left(m_e m_i\right)^{1/2} Z^2 n_e \lambda_{ei}} \tag{19}$$

where all parameters are in SI units, all temperatures are in Kelvin, k is Boltzmann's constant, Z is the charge state, n_e is the electron number density, λ_{ei} is the Coulomb logarithm, m_e is the electron mass, and m_i is the mass of the ion.

2.5 Equations of state

2.5.1 Ideal gas

2.5.2 Gruneisen equation of state

An equation of state which is analytic like ideal gas law, but more generally applicable, is the Gruneisen equation of state (e.g. see (Harlow and Amsden 1970) and (Addessio et al. 1986)). Ideal gas law has 1 parameter which sets it's behavior, that is the species. From that we get γ, MW, and R, and then the relationships are all determined. For Gruneisen, there are 3 parameters, ρ_0 (solid density of the material at 0 pressure and maybe room temperature), c (soundspeed of the unshocked material), and γ_s (Gruneisen ratio). Assuming that we know γ_s, c, and ρ_0, the Gruneisen equation of state, given pressure in terms of density ρ and internal energy e, is

$$p = p_H + \gamma_s \rho \left(e - e_H\right) \tag{20}$$

where p_H and e_H are the Hugoniot pressure and energy, respectively. Often they come from experimental data, and represent a point on a shock Hugoniot curve. If the soundspeed is known (often times it's obtained experimentally), then these two quantities are

$$p_H = \frac{c^2}{\left[1/\rho_0 - s\left(1/\rho_0 - 1/\rho\right)\right]^2} \left(\frac{1}{\rho_0} - \frac{1}{\rho}\right) \tag{21}$$

$$e_H = \frac{1}{2} p_H \left(1/\rho_0 - 1/\rho\right) \tag{22}$$

$$s = \frac{\left(\gamma_s + 1\right)}{2} \tag{23}$$

Also, it should be noted that

$$\gamma_s = \gamma - 1 \tag{24}$$

The temperature is given by

$$T = \frac{e - e_H}{C_v} + T_H \tag{25}$$

where T_H is the Hugoniot temperature, given by

$$T_H = e^{F+G\left(\ln\frac{1}{\rho}\right)+H\left(\ln\frac{1}{\rho}\right)^2+I\left(\ln\frac{1}{\rho}\right)^3+J\left(\ln\frac{1}{\rho}\right)^4}. \quad (26)$$

As far as I can tell, a reasonable approximation to this is to replace TH with the reference temperature of the material at which ρ_0 is determined, so basically something like room temperature. If you look up the density of a material, there will usually be a temperature provided at which this density was reported, so use that value in Kelvin. The soundspeed is given by

$$c^2 = \left(\frac{\partial p}{\partial \rho}\right)_e + \frac{p}{\rho^2}\left(\frac{\partial p}{\partial e}\right)_\rho \quad (27)$$

Using Eq. 20,

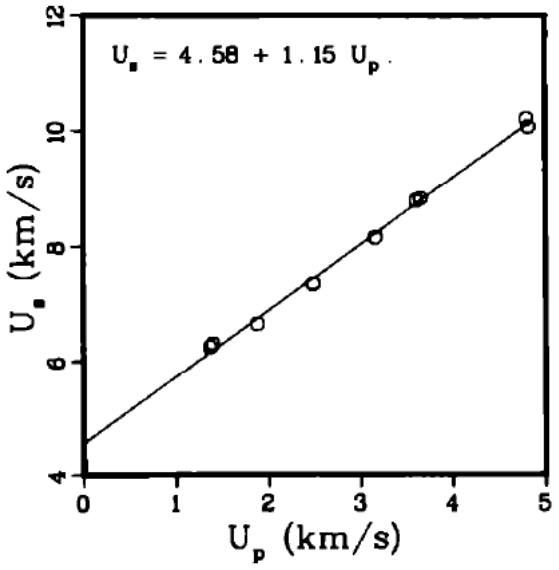
$$c^2 = \gamma_s (e - e_H) + \frac{p\gamma_s}{\rho} \quad (28)$$

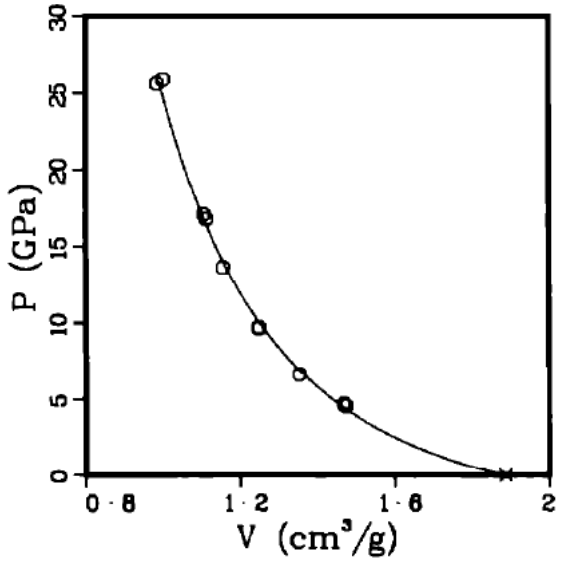
Okay, but how do we actually get the Gruneisen parameters? The constants c and s are constants obtained from experimental data in which a shock is driven through the material. The relationship between shock speed U_s and the particle speed U_p behind the shock, is given by

$$U_s = c + sU_p \quad (29)$$

Data are available for a wide variety of material. See for example (Marsh and Laboratory 1980).

For example, on p. 104 of Marsh et al, they have data for lithium with an initial solid density of $\rho_0 = 530$ kg/m^3. In this plot they already developed a linear fit, so that we can extract the soundspeed and s parameter at once, 4580 m/s and 1.15, respectively. Thus, the Gruneisen parameter $\gamma_s = 2s\text{-}1 = 1.30$. They also provide the actual data, along with corresponding measured pressure, so you can check your calculation of pressure. Keep in mind to evaluate the equations in SI units, and then convert from Pa to Gpa by dividing by 10^9 to compare with the measured data.





ρ_0 (g/cm³)	U_s (km/s)	U_p (km/s)	P (GPa)	V (cm³/g)	ρ (g/cm³)	V/Vo	Exp
.530	6.253	1.370	4.540	1.4734	.679	.781	im1 o
.530	6.306	1.401	4.682	1.4676	.681	.778	im1 o
.530	6.653	1.882	6.636	1.3531	.739	.717	im1 o
.530	7.360	2.487	9.701	1.2492	.800	.662	im1 o
.530	7.344	2.488	9.684	1.2476	.802	.661	im1 o
.530	8.152	3.159	13.649	1.1556	.865	.612	im1 o
.530	8.787	3.602	16.775	1.1134	.898	.590	im1 o
.530	8.822	3.650	17.066	1.1062	.904	.586	im1 o
.530	10.211	4.794	25.944	1.0010	.999	.531	im1 o
.530	10.070	4.814	25.693	.9848	1.015	.522	im1 o

2.5.3 Tabular equation of state

2.5.4 Modification for high density

The Gruneisen equation of state scales exactly like ideal gas law as a function of temperatures. One of the problems with this equation is that it incorrectly predicts states of tension beyond solid density. An improved model is the Anton-Schmidt equation, which provides a correction for van der Waal and other nonideal terms which depart from ideal gas law,

$$p(\rho) = -K_0\left(\frac{\rho_0}{\rho}\right)^n \ln\left(\frac{\rho_0}{\rho}\right) + p_0 \tag{30}$$

where K_0 is the bulk modulus and n is related to the specific heat ratio with

$$n = \frac{5}{6} - \gamma \tag{31}$$

and p_0 is the reference pressure at the reference density ρ_0, usually the solid density at 1 atmosphere. For higher temperatures, the pressure can be modeled as linear with temperature, so that p_0 can be replaced with ρRT and thus scales exactly as the Gruneisen equation of state. This equation gives negative pressure as the material undergoes tension, which is applicable until the yield stress is exceeded. We approximate the pressure for densities below that exceeding the yield stress as gradually with a numerical switch which smoothly transitions the equation of state back to perfect gas. The final modified equation is

$$p(\rho) = -K_0 \left(\frac{\rho_0}{\rho}\right)^n \ln\left(\frac{\rho_0}{\rho}\right)\left[\frac{1}{2} + \frac{1}{2}\tanh\left(m(\rho/\rho_0 - a)\right)\right]\left(T > T_{boil}\right) + \rho R T \tag{32}$$

where the term in the brackets is the switch function for the yield stress, and m and a are control parameters which scale width (in density) over which the switch is applied. Currently a = 0.8,

$$m = \frac{6}{b - a} \tag{33}$$

and b = 0.9. The inequality is a logical switch that also turns off this term in the case that the boiling temperature is exceeded. In summary, the inputs to this equation are K_0, ρ_0, γ, MW, and T_{boil}. When the pressure changes rapidly as it does in Eq. 32, the energy equation becomes very sensitive to density. The correction to linear calorically perfect gas theory is

$$\Delta e = \int_{\rho_0}^{\rho} \frac{\left[P(\rho) - \rho R T\right]}{\rho^2} d\rho \tag{34}$$

As an example, the following matlab code would evaluate this expression

```
rho = rho0*logspace(0,2,100000);
en = cumtrapz(rho,pfun(rho)./(rho.^2));
```

where `pfun` is a function handle to the numerator in Eq. 34.

2.6 Electromagnetic Field and Circuit Solver

The propagation of electromagnetic waves, current flow, and electromagnetic forces are determined by solving 1) a coupled set of transmission line equations through the conductors connected self consistently to external circuits and 2) Maxwell's equations through dielectric media and vacuum. Maxwell's equations are solved on two sets of staggered background points, one each for the electric and magnetic field

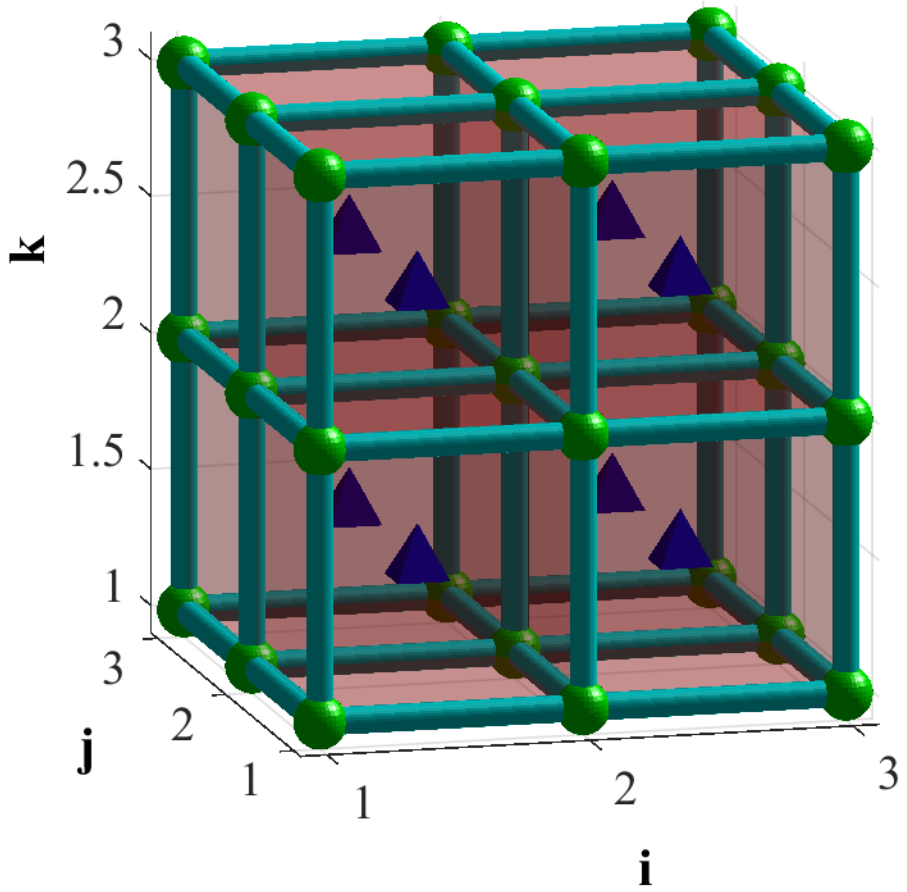


Figure 1. Representative nodes of electric (green circle) and magnetic (blue tetrahedra) field points.

2.6.1 Circuit Solver

The potential and current are solved self-consistently between an external circuit model a network of transmission line equations. To explain how the model works, we reference Fig. 2 below. The green circuit represents an example of an external circuit which drives current, voltage, electric, and magnetic fields inside the computational domain. The computational domain is represented by the red, gray, and black physical sph particle and the cyan (electric) and yellow (magnetic) field particles, where the electric and magnetic field points are staggered as shown in Fig. 1.

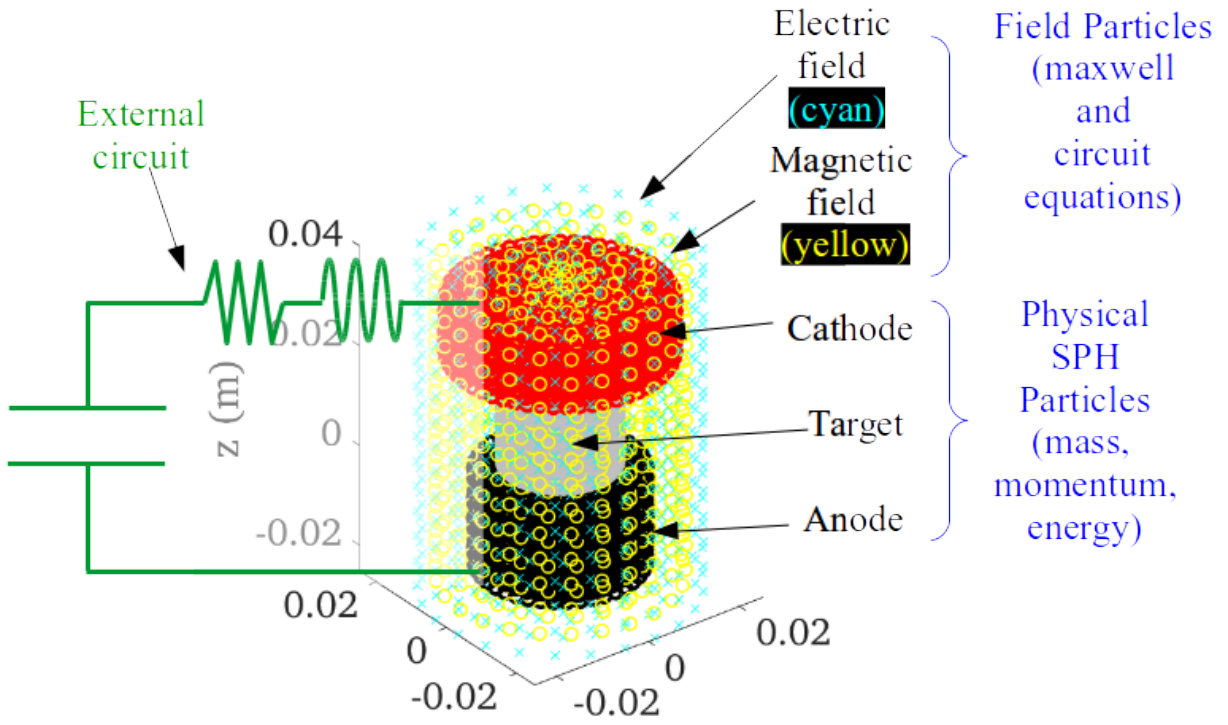


Figure 2. Example of external circuit and connections to anode and cathod of a z-pinch, with SPH particles (red, gray, and black) immersed in electric particles (cyan) and magnetic particles (yellow).

For an external circuit which can be modeled as a series of transmission line sections (including simple RLC circuits, marx banks, pulse forming networks, and linear transformer drivers), the equations are

$$
\begin{aligned}
\dot{V}_1 &= -\frac{I_1}{C_1} \\
\dot{I}_1 &= \frac{1}{L_1}\left(V_1 - R_1 I_1 - V_{T,1}\right) \\
\dot{V}_{T,1} &= \frac{l_T}{C_T}\left(\frac{I_1 - I_{T,1}}{\Delta z}\right) \\
\dot{V}_{T,i} &= \frac{l_T}{C_T}\left(\frac{I_{T,i-1} - I_{T,i}}{\Delta z}\right) \quad \text{i = 2 to N} \\
\dot{I}_{T,i} &= \frac{l_{T,i}}{L_{T,i}}\left(\frac{V_{T,i} - V_{T,i+1}}{\Delta z}\right) - \frac{I_{T,i} R_{T,i}}{L_{T,i}} \quad \text{i = 1 to N-1} \\
\dot{I}_{T,N} &= \frac{1}{L_{SPH}}\left(\phi_{SPH,pos} - \phi_{SPH,neg}\right) - \frac{I_{T,N} R_{T,N}}{L_{T,N}}
\end{aligned}
\tag{35}
$$

where the subscript T refers to a transmission line section. Note that the nodes N of the

transmission line could be capacitors with their own initial charging voltage, or could be bus bars or other passive elements transferring the current and voltage from the capacitor bank to the SPH particles. Note that the last equation, for the time rate of change of current, is directly connected to the SPH particles through the potential difference between the positive and negative SPH particles where the circuit is connected.

For all SPH particles, the transmission line equations for the physical SPH particles are given as

$$\frac{\partial \phi}{\partial t} = -\left(\frac{\mathbf{I} + \mathbf{G}\phi}{\mathbf{C_n}} \cdot \mathbf{n} \right) \tag{36}$$

where **n** is a unit normal for each face of the particle, the term on the RHS represents the net potential drop due to net rate of charge loss from the particle plus the conductance, where

$$G_n = \frac{\widetilde{A}\sigma}{\Delta n}, \; C_n = \frac{\widetilde{A}\varepsilon}{\Delta n} \text{ , n = x,y,z} \tag{37}$$

The current **I** in each particle is advanced in the x, y, and z directions with

$$\frac{\partial \mathbf{I_n}}{\partial t} = \frac{-\frac{\partial \phi_n}{\partial n} + \frac{\partial R}{\partial n}\mathbf{I_n}}{\frac{\partial L_n}{\partial n}} \text{ , n = x,y,z} \tag{38}$$

where

$$R_n = \frac{A_n}{\widetilde{\Delta} n \sigma}, \; L_n = \frac{A_n \mu}{\widetilde{\Delta} n} \tag{39}$$

Current density in each particle is given by

$$\mathbf{j_n} = \mathbf{I_n} / A_n \tag{40}$$

and charge density with

$$\frac{\partial \rho}{\partial t} = \frac{1}{vol} \mathbf{I} \cdot \mathbf{n} \tag{41}$$

where the I n term is the net current into the particle. In order to connect the external circuit to the SPH particles, the current from the external circuit is added to the positive and negative SPH particles where the circuit is connected. This will add potential to those boundary particles at a rate of I/C. Also, these boundary particles are set with a total current rise rate equal to $\dot{I}_{T,N}$ from the external circuit transmission line equations above.

2.6.2 Electromagnetic field solver

The way electric and magnetic fields are advanced depends whether the field particles are treated as conductors or vacuum/dielectric. The conductivity is determined by interpolated from the neighboring sph particles. For those which return a conductivity of 10^4 $(\Omega\text{-m})^{-1}$ or higher, the electric field is calculated with

$$\mathbf{E} = -\nabla \phi \tag{42}$$

and the magnetic field is computed with

$$\mathbf{B} = \nabla \times \mathbf{A} \tag{43}$$

For the field particles which reside in the vacuum in inside a dielectric media (as determined by interpolation from local SPH physical particles), Maxwell's equations are integrated in time with

$$\begin{aligned} \nabla \times \mathbf{H} &= \frac{\partial \mathbf{E}}{\partial \mathbf{t}} + \mathbf{J} \\ \nabla \times \mathbf{E} &= -\mu \frac{\partial \mathbf{H}}{\partial \mathbf{t}} \end{aligned} \tag{44}$$

where **H** represents the magnetic field, **E** – the electric field, **J** – the current density, and μ is the permeability of circuit material. Calculating changes in electric and magnetic fields is a bit different from calculating physical properties such as temperature and density in that changes depend on field strengths of each others neighboring particles. In other words, electric field changes depend on the strength of the magnetic field in neighboring particles, whereas magnetic field changes depend on the strength of the electric field in neighboring particles. This observation is more readily apparent in the expanded curl equations in the *x*, *y*, and *z* directions, where new variables $\varepsilon, \sigma,$ and V represent the permittivity, conductivity, and volume of the circuit material respectively.

$$\begin{aligned} \frac{\partial E_x}{\partial t} &= \frac{1}{\varepsilon} \sum_{j=1}^{N} \left(H_z\left(r_j^H\right) W_y^E - H_y\left(r_j^H\right) W_z^E \right) \Delta V_j + \frac{\sigma E_x\left(r_i^E\right)}{\varepsilon} - \frac{J_x}{\varepsilon} \\ \frac{\partial E_y}{\partial t} &= \frac{1}{\varepsilon} \sum_{j=1}^{N} \left(H_x\left(r_j^H\right) W_z^E - H_z\left(r_j^H\right) W_x^E \right) \Delta V_j + \frac{\sigma E_y\left(r_i^E\right)}{\varepsilon} - \frac{J_y}{\varepsilon} \\ \frac{\partial E_z}{\partial t} &= \frac{1}{\varepsilon} \sum_{j=1}^{N} \left(H_y\left(r_j^H\right) W_x^E - H_x\left(r_j^H\right) W_y^E \right) \Delta V_j + \frac{\sigma E_z\left(r_i^E\right)}{\varepsilon} - \frac{J_z}{\varepsilon} \end{aligned} \tag{45}$$

$$\begin{aligned} \frac{\partial H_x}{\partial t} &= \frac{-1}{\mu} \sum_{j=1}^{N} \left(E_z\left(r_j^E\right) W_y^H - E_y\left(r_j^E\right) W_z^H \right) \Delta V_j \\ \frac{\partial H_y}{\partial t} &= \frac{-1}{\mu} \sum_{j=1}^{N} \left(E_x\left(r_j^E\right) W_z^H - E_z\left(r_j^E\right) W_x^H \right) \Delta V_j \\ \frac{\partial H_z}{\partial t} &= \frac{-1}{\mu} \sum_{j=1}^{N} \left(E_y\left(r_j^E\right) W_x^H - E_x\left(r_j^E\right) W_y^H \right) \Delta V_j \end{aligned} \tag{46}$$

The expanded Maxwell's equations are based on formulations provided by Ala and Francomano with a slight difference in the additional current density term. This term becomes necessary to enable the SPFMax code to simulate interaction between the surface a solid electrode and the charged fluid particles of the fluid model (in this case, the fluid is a plasma). In the propulsion simulation, charged fluid particles take on current from the circuit model, essentially becoming part of it while simultaneously encompassing the fluid model as well. Certain dynamics of the circuit-fluid model are very different from the circuit-electrode model.

Because the electrode is a solid surface, the permittivity, permeability, and conductivity of the circuit material is assumed constant. However, those same properties must be interpolated in the plasma because the plasma is deforming and accelerating over the course of the simulation. As particle concentration changes, physical properties change.

2.6.3 Algorithm

1. The time rate of change of voltage and current are calculated for the external circuit elements
2. The time rate of change of potential and current density in the x, y, and z directions are computed with the transmission line equations for the material particles
3. The charge and current density are updated
4. The charge and current density are interpolated to the electric and magnetic field particles
5. The time rate of change of electric and magnetic fields are computed using the SPH variant of the finite difference time domain method
6. All time derivatives of SPH and field particle properties are integrated self-consistently

2.7 Ray tracing for electromagnetic radiation deposition, beam current deposition, fast ion deposition from fusion and fission, and fast fission reactions.

Note: See also Section3.3.5 Special Discussion about Rays to see at least one way of setting rays up.

Nonlocal deposition of energy from radiation sources is a challenging problem. The philosophy behind SPFMax is to implement algorithms which capture the physics accurately while maintaining the ability to run full 3D problems on a laptop or comparable desktop computer. The ray tracing algorithm implemented here is an attempt to handle nonlocal deposition with the ability to scale up as computers become more powerful. The basic approach is to specify a ray geometry and a type of source radiation (electromagnetic, fast ions from fusion or fission fragments, neutrons, or current), and the power of the radiation attenuates as it propagates through matter. There are exceptions to the attenuation, such as the specification of a current in a beam, which could be used to define a simple localized arc. Scattering is currently not modeled. The method assumes a single pass of the radiation from the source along the center axis of the ray or branch (see next paragraph for definition of rays and branches). We may explore including scattering in the future if it becomes important for problems of interest.

Collectively, here the term 'ray' means a beam (i.e. cylinder), axisymmetric cone, or 4π isotropic source of radiation in which the radiation/matter interaction occurs, as illustrated in Fig. 3. The intensity of the radiation source falls off as $1/r^2$ away from the source for the cone and 4π rays, but is collimated for the beam source. In any problem, the user is free to specify any number or combination of rays, although increasing the number and resolution of each ray will slow down

the computation. The 4π rays are unique among the 3 ray options in that they are split into a number of 'branches' specified by the user so that anisotropic material properties may interact with an isotropic source, so that heating or other interactions away from the source may vary in any direction, down to the resolution of the branches.

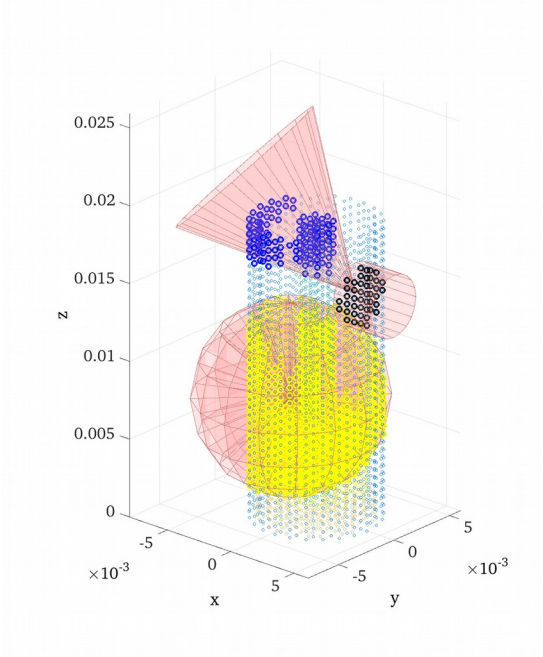


Figure 3. Options for ray tracing, showing a representative beam (cylinder), conical, and 4 pi (spherical) options. The black, blue, and yellow points are physical SPH particles which reside in each of the beam, cone, and 4pi rays, and are the particles which would interact with the radiation source

The present physics options for rays include ohmic dissipation, electromagnetic radiation (currently requires a Bremsstrahlung radiation to be turned on), fusion or fission fragment ion deposition, fast fission power generation (requires fusion reactions to be turned on). We discuss the physics models below.

2.7.1 Ohmic dissipation

The energy equation includes a term given by

$$\left(\frac{de}{dt}\right)_{ohmic} = \frac{\eta}{\rho} j^2 \quad \left[\frac{\mathrm{m}^2}{\mathrm{s}^3}\right] \tag{47}$$

Ohmic Power dropped in a ray segment is given by

$$P_{\Omega,r}=\sum_{b=1}^{N_{p,r}}\frac{\eta_{b,r}}{\rho_{b,r}}j_r^2 \tag{48}$$

where the subscript b,r means particle b in ray segment r, r means ray segment r, and $N_{p,r}$ is the number of particles in ray segment. Note the current is still set up with the circuit. Basically what setting up a ray does is limit the current to only dissipate energy inside the beam and nowhere else.

2.7.2 Bremsstrahlung radiation power per unit mass

For electromagnetic radiation, if the Bremsstrahlung option is turned on, then the radiation power added to the energy equation is

$$\begin{aligned}\left(\frac{de}{dt}\right)_{br} &= \frac{-16\pi}{3\sqrt{6\pi}}\frac{q^6}{m_e^2c^3(4\pi\varepsilon_0)^3}\frac{Z_i^2 n_e}{\sqrt{k_B T_e/m_e}MW_i m_a}\int_0^\infty 4\pi\exp\frac{-h\nu}{k_B T_e}d\nu \\ &= \frac{-64\pi^2}{3\sqrt{6\pi}}\frac{q^6}{hm_e^2c^3(4\pi\varepsilon_0)^3}\sqrt{\frac{k_B T_e}{m_e}}\frac{Z_i^3\rho}{(MW_i m_a)^2}\end{aligned} \tag{49}$$

For spectrally dependent (i.e. multigroup) Bremsstrahlung radiation,

$$\left(\frac{de}{dt}\right)_{br_\nu}=\frac{-64\pi^2}{3\sqrt{6\pi}}\frac{q^6}{m_e^2c^3(4\pi\varepsilon_0)^3}\sqrt{\frac{k_B T_e}{m_e}}\frac{Z_i^3\rho}{(MW_i m_a)^2}\left(e^{-\frac{h\nu_{i+1}}{k_B T_e}}-e^{-\frac{h\nu_i}{k_B T_e}}\right) \tag{50}$$

The Bremsstrahlung absorption coefficient (free free opacity), and works exactly like the mass attenuation coefficient

$$\kappa_\nu=\frac{8\pi}{3\sqrt{6\pi}}\frac{q^6}{hm_e^2c(4\pi\varepsilon_0)^3}\frac{Z_i^3\rho}{(MW_i m_a)^2\nu^3}\sqrt{\frac{m_e}{k_B T_e}}=\frac{\mu}{\rho}\quad\left[\frac{m^2}{kg}\right] \tag{51}$$

As a side note, the NIST mass attenuation tables give a parameter (μ/ρ) which is the same thing as κ_ν.

Finally, the Bremsstrahlung Power dropped in a ray segment is given by

$$P_{\nu,r}=P_{in,\nu,r}\left(1-e^{-\kappa_\nu\rho\Delta r}\right) \tag{52}$$

where the power distributed in each ray segment is given by

$$P_{\nu,0r}=\left(\frac{de}{dt}\right)_{br_\nu}\frac{\Omega_r}{4\pi} \tag{53}$$

and the solid angle subtended by the ray is

$$\Omega_r=\sin\phi\,d\phi\,d\theta \tag{54}$$

where φ is the angle from the z-axis (polar angle) and θ is the azimuthal angle measured CCW

from the x-axis around the z-axis.

2.7.3 Fusion power

The reactivity for a reaction of species i with species j is given by

$$\frac{dN}{dt}=\frac{n_{X_i}n_{X_j}}{1+\delta_{ij}}<\sigma v>_{ij}\mathcal{V} \tag{55}$$

where $\delta_{ij}=\begin{cases}1 & i=j\\ 0 & i\neq j\end{cases}$. The power per unit mass delivered to a particular fusion product specie k is

$$\left(\frac{de}{dt}\right)_k=\frac{n_{X_i}n_{X_j}}{\rho\left(1+\delta_{ij}\right)}<\sigma v>_{ij}E_k \tag{56}$$

where E_k is the initial kinetic energy of the fusion product specie k. The total power of the system is

$$P_{fus,k}=\sum_{a=1}^{N}\frac{n_{a,X_i}n_{a,X_j}}{\left(1+\delta_{ij}\right)}<\sigma v>_{a,ij}E_k\mathcal{V}_a \tag{57}$$

where the sum is over all SPH particles. The fusion power deposited in a specific ray segment is then calculated with

$$P_{ion,0r}=\left(\frac{n_{X_i}n_{X_j}}{1+\delta_{ij}}<\sigma v>_{ij}\mathcal{V}\right)\left(\frac{\Omega_r}{4\pi}\right)\left(\left(\frac{dE}{dr}\right)_{ir}\Delta r_{ir}\right) \tag{58}$$

The second term on the RHS accounts for the solid angle subtended by the ray segment, Ω_r . The last term is work done by a fast fusion ion traveling through a particular segment *i* of a particular ray where $(dE/dr)_{ir}$ is the function giving the local stopping power (deceleration force) on the fast particles determined as a function of ion species and electron densities and temperatures. The local stopping power itself is a function of the kinetic energy of the fast ions moving through the ray segment, $0.5m_k v_{k,ir}^2$. It is assumed that through a ray segment, the stopping power is constant. The total change of kinetic energy of the ion in the segment is then

$$\frac{1}{2}m_k v_{k,ir}^2-\frac{1}{2}m_k v_{k,i(r-1)}^2=-\delta W_{ir} \tag{59}$$

The velocity is then

$$v_{k,ir}=\max\left[\left(v_{k,i(r-1)}^2-\frac{2\delta W_{ir}}{m_k}\right),0\right] \tag{60}$$

The power is then distributed to the particles in the ray segment based on the compact support size h of each particle. The actual power per unit mass deposited in an sph particle ‘a’ is then

$$\left(\frac{de}{dt}\right)_a = \sum_{ir=1}^{\text{number of rays}} \left[\frac{h_a}{m_a}\left(\frac{n_{X_i} n_{X_j}}{1+\delta_{ij}} <\sigma v>_{ij} \mathcal{V}\right)_{ir} \left(\frac{\Omega_r}{4\pi}\right)_a \left(\left(\frac{dE}{dr}\right)_{ir}\right)\right] \tag{61}$$

where m_a is the mass of the particle. There are currently 2 available stopping power models:

Bethe model
%inputs
Za %charge of fast ion
Ma %Atomic weight of fast ionization in amu
Ea0 %born on energy in SI for fast ion
Zk %background plasma
nk %number density of ion background, or mass density

$$\frac{dE}{dx} = -\frac{q^4}{4\pi\varepsilon_0^2}\frac{Z_a^2 n_e}{m_e v_a^2}\ln\left(\frac{4\pi m_a v_a^2}{h\omega_{pe}}\right) \tag{62}$$

Harris Miley
%inputs
Za %charge of fast ion
Ma %Atomic weight of fast ionization in amu
Ea0 %born on energy in SI for fast ion
Zk %background plasma
Mk %atomic weight of background species
nk %number density of ion background, or mass density
Tk %background plasma temperature to get the thermal speed
lnlambda %scalar value we can set as a function later

The force on an energetic charged particle produced by fusion reactions is

$$\frac{dE_\alpha}{dx} = -\frac{2\pi Z_\alpha^2 Z_k^2 q^4 n_k m_\alpha \ln\lambda}{(4\pi\varepsilon_0)^2 m_k E_\alpha} F\left(\frac{V_\alpha}{V_{k,th}}\right) \tag{63}$$

where

$$F(x) = \text{erf}(x) - \frac{2}{\pi^{1/2}}\left(1+\frac{m_k}{m_\alpha}\right) x e^{-x^2} \tag{64}$$

where

$$V_{k,th} = \left(\frac{2kT_k}{m_k}\right)^{1/2} \tag{65}$$

The power dissipated in the local ray section is then given by

$$P_{dep,ir} = \frac{dN}{dt} f_r \delta W_{ir} \tag{66}$$

2.7.4 Fission power

The fission power is determined assuming a single pass of fast neutrons from fusion source through a surrounding cold liner, and is given for a ray segment i of a particular ray r as

$$P_{fis} = \left(\frac{d N}{dt} \right)_{fis} f_{ir} Q_{fis} \tag{67}$$

where n_{ir} is number density of the fission fuel reacting with the neutrons, σ_{fis}, is the fast fission cross section which needs to be a function of the energy of the neutron from the fusion reactions and the species it is reacting with, and Q_{fis} is the energy released per reaction. As a reminder, f_r is the fractional power in a ray, based on the solid angle subtended by the segment.

For reference I'll include the derivation. The fission reactivity in a ray segment is given by

$$\left(\frac{dN}{dt} \right)_{fis,ir} = n_{fis} n_{neutrons} \sigma_{fis} v_{neutron} \mathcal{V}_{ir} \tag{68}$$

The local neutron number density in a ray segment coming from an isotropic source to r_{ir} (neglecting scattering) from the the center is given by (assuming the shell thickness at r_{ir} is small enough for the approximation made to the spherical shell volume)

$$n_{neutron,ir} = \frac{N}{\mathcal{V}_{ir}} = \left(\frac{dN}{dt} \right)_{neutrons,ir} \frac{1}{4 \pi r_{ir}^2 v_{neutron}} \tag{69}$$

because $v_{neutrons} = dr/dt$. So the fission reactivity becomes

$$\left(\frac{dN}{dt} \right)_{fis,ir} = n_{fis} \sigma_{fis} v_{neutron} \mathcal{V}_{ir} \left(\frac{dN}{dt} \right)_{neutrons,ir} \frac{1}{4 \pi r_{ir}^2 v_{neutron}} \tag{70}$$

which simplifies to

$$\left(\frac{dN}{dt} \right)_{fis,ir} = \left(\frac{dN}{dt} \right)_{neutrons,ir} n_{fis} \sigma_{fis} \Delta r_{ir} \tag{71}$$

The neutron reaction rate has to account for fusion and fission neutrons, and is nonlinear because the fission neutron production rate amplifies itself. Since one has to sum from all the sources, it's more convenient to pull the total neutron production rate out, and multiply it by the solid angle subtended by the ray segment, and sum over all the ray segment specific terms, multiplying by the total neutron production of the source acting on the entire ray,

$$\left(\frac{dN}{dt} \right)_{neutrons} = \left(\frac{dN}{dt} \right)_{fus} + \left(\frac{dN}{dt} \right)_{neutrons} \sum \left(n_{fis} \sigma_{fis} \Delta r_{ir} v_{neutrons} f_{ir} \right) \tag{72}$$

where fis, fus, sph refer to fission, fusion, and smooth particle hydrodynamic particle, respectively. $v_{neutron}$ is the neutron multiplier and is typically a number like 2.7 or so. Solving this expression for the rate of change of the number of neutrons

$$\left(\frac{dN}{dt}\right)_{neutrons} = \left(\frac{dN}{dt}\right)_{fus} \times \left[\frac{1}{1-\sum\left(n_{fis}\sigma_{fis}\Delta r_{ir}\nu_{neutrons}\right)}\right] \tag{73}$$

In general, the fusion neutrons could be replaced with, or added to, other neutron sources. The way this equation is implemented would be to calculation the fission number density in each ray segment, and then limit the denominator so it is never less than .01 (negative, or cause a singularity). Some problems with these models is that they don't factor in finite burnup of the fuel. We need to revisit that if it's important.

$$P_{fis} = \left(\frac{d\,N}{dt}\right)_{neutrons} \sum\left(f_r n_{ir}\sigma_{fis}\Delta r_{ir} Q_{fis}\right) \tag{74}$$

2.7.5 To update SPFMax ray tracing, you need to edit the following

define_equations_of_motion – add a source term to the source equations (for energy)
define_physics_models -
define_transport_models -
set_input_defaults – for any new variables introduced in problem setup, or to add additional comments for new options to existing inputs variables
set_rayitem_defaults – set up physics and conditions of ray geometry

The source terms are currently listed below, and each one needs an 'if physics.rays' block to tell it to connect the power to the ray tracing algorithm. Inside each of these if blocks it also needs one or more of the following set to 1 or 'true' rayitem(ii).use_current_density, rayitem(ii).use_neutron, rayitem(ii).use_ion, or rayitem(ii).use_em_radiation', 0);

1. *energy_ohmicheating* – ohmic heating
2. *radiation_thin* – optically thin radiation
3. *radiation_Bremsstrahlung* – spectrally dependent Bremsstrahlung radiation option
4. *fusion_power* – charged particle and neutron products
5. *fission_power* – products from fission power

3. Setting up Problems

3.1 Overview

Problems are currently generated by 1) creating the geometry either by a GUI (/geometry/GUICreateBlockStart) or matlab script referred to as the geometry file, and 2) create the input file. If GUICreateBlockStart is not used, the geometry file is run independently and creates .mat files for the geometry of the physical items, like blobs of gas, walls, electrodes, insulators, humans (we haven't created a human yet, but might be fun), etc, which must be

described by a discrete set of points with x, y, and z coordinates. They have to be in 3D. Details are discussed in the next section.

Input files are matlab scripts which are run from within spfmax. They read the geometry file, set up initial conditions, physics models, and solver parameters. More details to follow.

3.1.1 What time is it in the simulation?

obj.time is the current simulation time. It goes from obj.time to obj.time + obj.dt up to 3 times. obj.dt is determined by the particle size h divided by the soundspeed plus the relative particle speed, or solver.max_time_step. the a Think of obj.dt as a global integration time step for all physics, but the local time step taken is always smaller than the global time step inside a loop. The time is advanced and reset for 3 sets of physics:

1. if Maxwell_solver is used, the time step is based on the electromagnetic wave speed from particle to particle, and is often 1/1000 or so of obj.dt. Performed in @particle/maxwell_solver
2. if hydro is used, the time step is obj.dt/4. Performed in @particle/RK.m
3. if source equations are used (like radiation, fusion, heat conduction), it is based on the heat conduction time step determined in source_timestep. Performed in @particle/RK.m after the hydro loop.

After the Maxwell, the time is obj.time + obj.dt. But then the time is rewound back to obj.time. Then hydro advances the time again to obj.time + obj.dt in RK2.m. Then the time is rewound again back to obj.time. The source equations again integrate the time to obj.time+obj.dt. The RK2 is exited and the code gets back to run_sph_solver. The code then checks to see if the output time has been exceeded, to determine if an output file needs to be saved.

3.1.2 Controlling the time step

upate_time_step is called in run_sph_solver before calling RK2 where the equations of motion are integrated. update_time_step determines the value of obj.dt which is the time step used hydro, hydro_Lorentz, energy, energy_i, energy_e, get_artificial_viscosity, hydro_viscous.

For heat conduction, radiation and other transport physics, the time step is further limited.

3.1.2.1 Forcing an absolute maximum time step

In the input file, set

solver.max_time_step = <your value here>

As the code proceeds, it will not exceed this value as the maximum time step taken

3.1.2.2 Setting the output time for post processing

In the input file, set

output.time_step = <your value here>

The output files will be generated approximately every time step set by your value. It will not be exact because that bottlenecks the computational time. If you need exact output times we can change that in 1 line of code, but if you are my student there is a better way than forcing the output time.

3.2 Setting up Geometry files

SPFMax has a GUI (/geometry/GUICreateBlockStart) which creates blocks and writes the appropriate scripts for the input files. You can also do this manually.

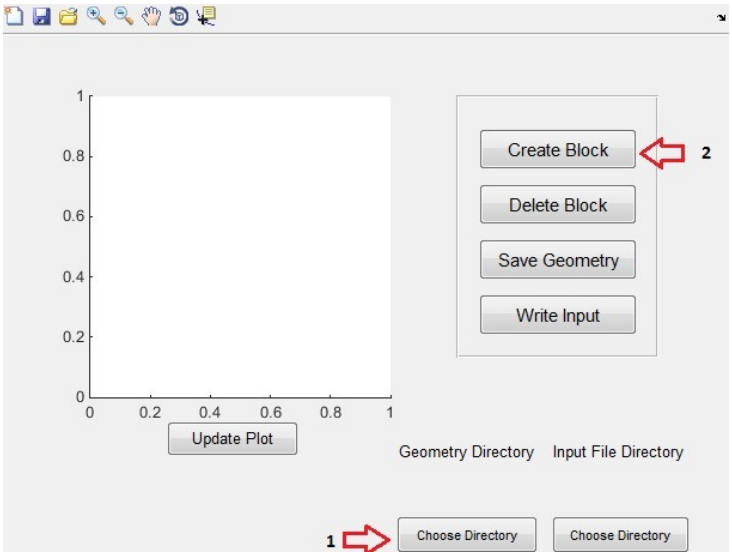
Geometry files are the preferred way to create the initial particle distribution, which is then read by the input file which then sets up what physics, numerical settings, solver settings, initial conditions, boundary conditions are to be set to define the problem. A number of examples are located in SPFMax/geometry. The geometry files are matlab scripts that can be organized into a few basic steps that we name as follows: directories, set up of blocks, plotting results, saving results.

3.2.1 GUICreateBlockStart: the GUI for creating the geometry (by James Patton)

The objective of this section is to illustrate how to use GUICreateBlockStart and geometry_selection to create geometries and input files.

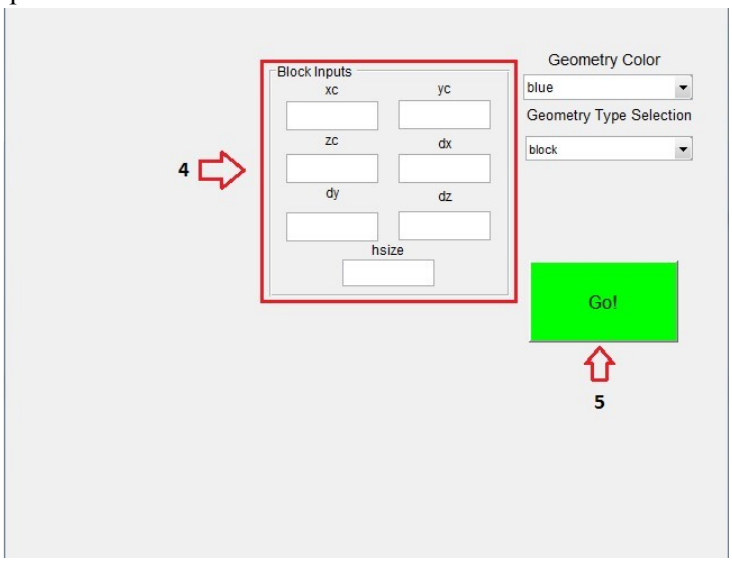
*3.2.1.1 **Creating geometry files:***

1. All the files needed to create geometries are in the same subdirectory of SPFMax, /geometry (i.e. shape functions, distance functions, make_block, CreateBlock, and geometry_selection)
2. From the /geometry directory, run GUICreateBlockStart.m
3. Press both push buttons labeled “choose directory” (lower right in figure below). These allow the user to select the directory for the geometry file and the directory for the input file.
4. The next step is to press the pushbutton labeled “Create Block.” This will open geometry_selection.m

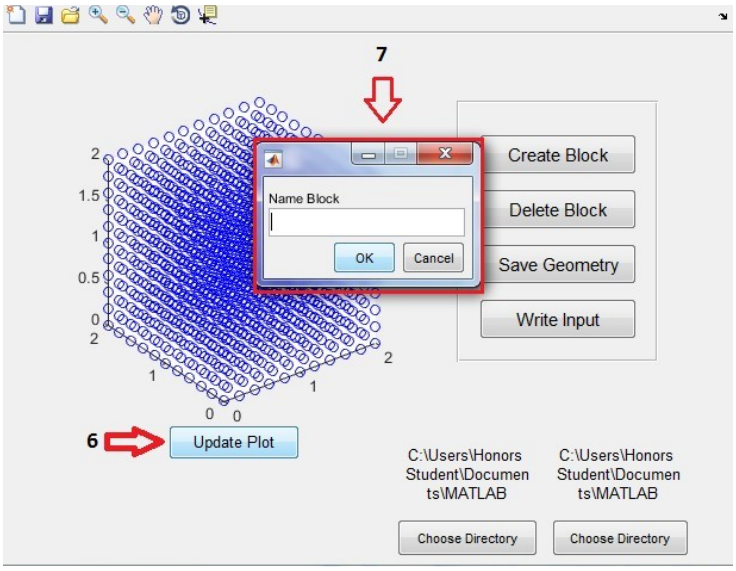


5. After this, a new dialogue box opens up (see figure below). Specify the geometry type and color, using the drop down menus labeled “Geometry Color” and “Geometry Type Selection.”

6. The next step is to fill out the inputs for the selected geometry type (in this manual "Block" will be used as an example).
7. Then press "Go!"



8. Next, go back to the original GUI and press the "Update Plot" pushbutton; then name the geometry.



9. To create more geometries, **repeat steps 3-7**

3.2.1.2 Deleting geometries

Press the "Delete Block" push button. Each geometry is assigned a number 1-n (n = total number of blocks). Input the number that corresponds to the geometry to delete it. Once a geometry is deleted, the geometries that are assigned a number higher than that of the geometry deleted, have their number incremented down by 1.

3.2.1.3 Saving geometries

Press the "Save Geometry" push button and select a directory.

3.2.1.4 Writing input

Once you have saved the geometries, click on the 'write input' and follow the prompt for saving the script. Then go to the directory where the file is saved, open it, and cut and paste the lines of code into your input file. Tada!

3.2.2 Creating geometries manually

3.2.2.1 Step 1, directories

initial statements to make function calls work and location for saving the output geometry. The only thing you need to change is the output_directory, to suit a subfolder location for the files. They don't have to go to SPFMax/geometry/<your subfolder>. It's just convenient for keeping them organized. Throughout, we use the example rail_shock_v7.m which captures most of the features we use.

```
close all
clear all
clc
home = pwd;
addpath([pwd '/distmesh']);

if exist('geometry')
   clear geometry
end
```

```
output_directory = 'rail_shock_dir_v7';
```

3.2.2.2 Step 2, set up of blocks

We have explored a lot of ways of creating the blocks. What we have settled on is the creation of a structured array called 'input' which must store the following: distance_function_str, distance_function_path, call to distance. The first part is to establish relevant scale lengths pertaining to this problem. You can call them whatever you want, but keep the term 'hsize'. This basically sets your particle spacing (approximately).

```
%inner conductor
scale_length = 0.012;
hsize = scale_length/4;
rail_length = 0.3;
```

Next a structured array inputs is created. The fields of the structure depend on the distance function used to create the geometry. (We'll get to distance functions next, so hang tight). What is common to all inputs fields are distmesh_function, distance_function_str, distance_function_path, distance_function, name, and hsize. The rest depend on the distance function called. After you set the fields for the input, you call get_geometry which then takes the inputs as the input to generate the particle geometry. The geometry output from get_geometry is a structured array, and this is the actual data saved to a file and read by SPFMax. See below the 8 blocks created by setting up inputs, numbering them sequentially 1, 2, etc, and then saving them to geometry(1), geometry(2), etc.

```
inputs(1).name = 'x1insulator';
inputs(1).xc = -scale_length/2 - hsize;
inputs(1).yc = -scale_length/2;
inputs(1).zc = 0;
inputs(1).dx = hsize;
inputs(1).dy = scale_length;
inputs(1).dz = rail_length;
inputs(1).hsize = hsize;
inputs(1).distmesh_function = 'block';
inputs(1).distance_function_str = 'distancefunction_block';
inputs(1).distance_function_path = pwd;
inputs(1).distance_function = distancefunction_block(inputs(1));
geometry(1) = get_geometry(inputs(1));

inputs(2).name = 'x2insulator';
inputs(2).xc = scale_length/2;
inputs(2).yc = -scale_length/2;
inputs(2).zc = 0;
inputs(2).dx = hsize;
inputs(2).dy = scale_length;%get dynamic fluid particle neighbor list
inputs(2).dz = rail_length;
inputs(2).hsize = hsize;
inputs(2).distmesh_function = 'block';
inputs(2).distance_function_str = 'distancefunction_block';
inputs(2).distance_function_path = pwd;
inputs(2).distance_function = distancefunction_block(inputs(2));
geometry(2) = get_geometry(inputs(2));

inputs(3).name = 'y1rail';
inputs(3).xc = -scale_length/2 - hsize;
inputs(3).yc = -scale_length/2 - hsize;
```

```
inputs(3).zc = 0;
inputs(3).dx = scale_length+2*hsize;
inputs(3).dy = hsize;
inputs(3).dz = rail_length;
inputs(3).hsize = hsize;
inputs(3).distmesh_function = 'block';
inputs(3).distance_function_str = 'distancefunction_block';
inputs(3).distance_function_path = pwd;
inputs(3).distance_function = distancefunction_block(inputs(3));
geometry(3) = get_geometry(inputs(3));

inputs(4).name = 'y2rail';
inputs(4).xc = -scale_length/2 - hsize;
inputs(4).yc = scale_length/2;
inputs(4).zc = 0;
inputs(4).dx = scale_length+2*hsize;
inputs(4).dy = hsize;
inputs(4).dz = rail_length;
inputs(4).hsize = hsize;
inputs(4).distmesh_function = 'block';
inputs(4).distance_function_str = 'distancefunction_block';
inputs(4).distance_function_path = pwd;
inputs(4).distance_function = distancefunction_block(inputs(4));
geometry(4) = get_geometry(inputs(4));

inputs(5).name = 'ar_gas1'; f1 = 1;
inputs(5).xc = -scale_length/2+hsize*f1;
inputs(5).yc = -scale_length/2+hsize*f1;
inputs(5).zc = hsize*f1;
inputs(5).dx = scale_length - 2*hsize*f1;
inputs(5).dy = scale_length - 2*hsize*f1;
inputs(5).dz = rail_length/2 - hsize*f1;
inputs(5).hsize = hsize*f1;
inputs(5).distmesh_function = 'block';
inputs(5).distance_function_str = 'distancefunction_block';
inputs(5).distance_function_path = pwd;
inputs(5).distance_function = distancefunction_block(inputs(5));
geometry(5) = get_geometry(inputs(5));

inputs(6).name = 'ar_gas2'; f2 = 1;
inputs(6).xc = inputs(5).xc;
inputs(6).yc = inputs(5).yc;
inputs(6).zc = inputs(5).zc+inputs(5).dz+0.5*(f1+f2)*hsize;
inputs(6).dx = inputs(5).dx;
inputs(6).dy = inputs(5).dy;
inputs(6).dz = inputs(5).dz;
inputs(6).hsize = hsize*f2;
inputs(6).distmesh_function = 'block';
inputs(6).distance_function_str = 'distancefunction_block';
inputs(6).distance_function_path = pwd;
inputs(6).distance_function = distancefunction_block(inputs(6));
geometry(6) = get_geometry(inputs(6));


inputs(7).name = 'z1breech';
inputs(7).xc = -scale_length/2-hsize;
inputs(7).yc = -scale_length/2-hsize;
inputs(7).zc = -hsize;
inputs(7).dx = scale_length+2*hsize;
inputs(7).dy = scale_length+2*hsize;
```

```
inputs(7).dz = hsize;
inputs(7).hsize = hsize;
inputs(7).distmesh_function = 'block';
inputs(7).distance_function_str = 'distancefunction_block';
inputs(7).distance_function_path = pwd;
inputs(7).distance_function = distancefunction_block(inputs(7));
geometry(7) = get_geometry(inputs(7));

inputs(8).name = 'z2breech';
inputs(8).xc = inputs(7).xc;
inputs(8).yc = inputs(7).yc;
inputs(8).zc = inputs(6).hsize+inputs(6).zc + inputs(6).dz;
inputs(8).dx = inputs(7).dx;
inputs(8).dy = inputs(7).dy;
inputs(8).dz = inputs(7).dz;
inputs(8).hsize = hsize;
inputs(8).distmesh_function = 'block';
inputs(8).distance_function_str = 'distancefunction_block';
inputs(8).distance_function_path = pwd;
inputs(8).distance_function = distancefunction_block(inputs(8));
geometry(8) = get_geometry(inputs(8));
```

Each geometry array stores the following fields

```
      x
      y
      z
   volume
      h
      N
      p: [1010x3 double]
      t: [2400x4 delaunayTriangulation]
   inputs: [1x1 struct]
```

x, y, and z are vectors of the x, y, and z points of the particle locations. volume is an approximate volume of the block, which is not important any more. h is the particle spacing and is also not as critical as it used to be. N are the number of points. p is an Nx 3 matrix storing x, y, z, and is only used for the distance function. t is a delaunay Triangulation of the points, also not important any more. Finally, inputs is the corresponding inputs structured array pertaining to this block. This last output is as important as the x, y, z. In summary, geometry stores everything needed to reproduce the geometry plus some extra information that SPFMax doesn't have to have but can save time. **The absolute necessary information stored in geometry is x, y, z, and inputs.**

3.2.2.3 Understanding distance functions

This is the hardest part about setting up geometries. The distance functions available are currently

distancefunction_block.m

distancefunction_coil_with_leads.m

distancefunction_cylinder.m

distancefunction_cylindrical_shell.m

distancefunction_spherical_shell.m

distancefunction_rail.m

What these functions do is dynamically create a handle to a function in which a surface is described implicitly by F(x,y,z) = 0. If you are in the interior of the surface, the function is negative and magnitude of F is roughly equal to the distance to the nearest part of the surface. When you call distancefunction_block with 'inputs' structured array, what it does is compute the parameters of the distance function based on the fields of the inputs array, and return a handle to the distance function. Basically it is writing a distance function for you. The handle is then used by spfmax later on to help particles figure out how close they are to walls.

To make a new distance function, start with an old one. Save it as distancefunction_blahblahwhatever, and then change the functions and inputs such that fd=distancefunction(p) returns a value of negative if inside the object, 0 on the surface, and positive outside of it. It is hard to learn how to do. The argument p for fd = distancefunction (p) are the x, y, z coordinates passed to the function and fd is the numerical value returned. The technique I use to create a new one is to set up the desired input fields you need to fix the geometry of the object you want, then hard code some test points that you know will be on the surface, out of th surface, and in the surface into the distance function, then call it, and plot fd vs the points along x, y, or z perhaps to make sure it is negative, 0, and positive in the right places.

3.3 Input files

Input files are made after the blocks are generated in the previous step. The easiest way to get started is to open up an existing input file with physics close to yours and go from there. The basics are to set up the name `run_name` and the structured arrays `matter`, `item`, `solver`, `physics`, and `circuit`. The most important ones are matter and item. They define the material properties and initial conditions of the blocks. Each element of material defines a new material. Each element of item defines a block.

3.3.1 Material properties

All materials (gases, plasmas, solids, liquids) have their properties set in the input file. You set the values of the properties in a structured array called 'matter'. For example, say you had hydrogen and copper in your problem. Your input file might look like this:

```
%------------------materials
matter(1).name = 'H2';
matter(1).phase = 2;
matter(1).eos = 'idealgas';  %can be tabular
matter(1).MW = 2.505;  %kj/kmol
matter(1).g = 5/3; %specific heat ratio
matter(1).chemistry = 'frozen';  %or 'nonequilibrium'
matter(1).conductivity = 1e4;  %nice plasma conductivity (gives diffusivity of 80)
```

```
matter(1).thermal_conductivity = 100;

matter(2).name = 'copper';
matter(2).phase = 'ablation';
matter(2).T_melting = 1356.15;
matter(2).T_boiling = 2840.15;
matter(2).relative_permittivity = 1;
matter(2).relative_permeability = 1;
matter(2).eos = 'solid';    % EOS Solid change
matter(2).conductivity = 5.95e7 ;  %1/ohm-m
matter(2).hydrostatic = 'stationary';
matter(2).thermal_conductivity = 401;    %IF THIS SHOWS 1, REMEMBER TO CHANGE THIS
LATER
matter(2).density = 8960;
matter(2).specific_heat = 386;
```

You don't have to specify everything. Most properties have a default value as shown below, and see 'set_matter_defaults' for the latest update. If some properties do not have a default value, we can add them in. I just overlooked doing so.

```
   matter = set_default_if_needed(matter,ii,'name','');
   matter = set_default_if_needed(matter,ii,'eos','none');
   matter = set_default_if_needed(matter,ii,'Z','none');  %ionization state.
   matter = set_default_if_needed(matter,ii,'hydrostatic','');  %this turns equation of motion
on or off
   matter = set_default_if_needed(matter,ii,'MW',28.97);
   matter = set_default_if_needed(matter,ii,'g',1.4);
   matter = set_default_if_needed(matter,ii,'chemistry','frozen');
   matter = set_default_if_needed(matter,ii,'relative_permittivity',1);
   matter = set_default_if_needed(matter,ii,'relative_permeability',1);
   matter = set_default_if_needed(matter,ii,'hydrostatic','');
   matter = set_default_if_needed(matter,ii,'conductivity',0); %1/ohm-m
   matter = set_default_if_needed(matter,ii,'viscosity',0); %Pa-s
   matter = set_default_if_needed(matter,ii,'thermal_conductivity',0); %W/m-K
   matter = set_default_if_needed(matter,ii,'phase',2); %gas  0 = solid, 1=liquid
    matter = set_default_if_needed(matter,ii,'T_melting',273.15); %melting point of ice at 1
atm
   matter = set_default_if_needed(matter,ii,'T_boiling',373.15); %boiling point of water at 1
atm
   matter = set_default_if_needed(matter,ii,'radiation','none'); %diffusion, thin, edny
      matter = set_default_if_needed(matter,ii,'opacity_Planck_abs',0);  %set a value, or
Bremmstrahlung, or tabular
     matter = set_default_if_needed(matter,ii,'opacity_Planck_emis',0);   %set a value, or
Bremmstrahlung, or tabular
      matter = set_default_if_needed(matter,ii,'opacity_Rosseland',0);    %set a value, or
Bremmstrahlung, or tabular
```

3.3.2 Setting Equation of State

Determines the relationship between pressure, density, and temperature. Also determines formula for relating temperature to specific internal energy. Options include ideal gas, tabular, gruneisen, solid, liquid. For an equation of state for handling a shock, compression, or expansion of a solid or liquid, there is another parameter called physics.compressibility_correction. See section 3.3.3.1 for an example.

3.3.2.1 Ideal gas

matter(1).eos = 'idealgas';
matter(1).MW = 39.948; %molecular or atomic weight, kg/kmol
matter(1).g = 1.3; %specific heat ratio
matter(1).chemistry = 'frozen'; %this is the default, some day maybe we'll do nonequilibrium

$$P = \rho R T \tag{75}$$

$$e = C_v T \tag{76}$$

$$C_v = \frac{R}{(\gamma - 1)} \tag{77}$$

3.3.2.2 Tabular

matter(1).eos = 'tabular'
matter(1).name = <your string here>;

The name is any .mat file in the material_database subdirectory. We can make more if what you need is not there.

3.3.2.3 Liquid

3.3.2.4 Solid

3.3.2.5 Gruneisen

3.3.3 Equation of state recommendations for hard problems

3.3.3.1 compression/expansion of a solid

matter(ii).name = <your material name here>

matter(ii).eos = 'idealgas';

matter(ii).relative_permittivity = 1;

matter(ii).relative_permeability = 1;

matter(ii).conductivity = 3.57e7; %1/ohm-m

matter(ii).thermal_conductivity = 27.5; % [W/mK]

matter(ii).MW = 235; %kg/kmol

matter(ii).g = 1.67;

matter(ii).opacity_Planck_abs = 'nist';

matter(ii).Z = 1;

matter(ii).T_boiling = 3505;

matter(ii).bulk_modulus = 100e9;

matter(ii).solid_density = 18900;

physics.compressibility_correction = 1; %turn on compressibility to account for compression of solids, needs a lot of inputs (solid_density, bulk_modulus, boiling temperature, all set in matter, and can be material dependent)

3.3.4 Turning on physics models

3.3.4.1 Hydrodynamics

physics.hydrodymamics = 1;. Leave it this way. Hydro is used for everything.

3.3.4.2 Hydrostatics

Controls whether or not particles are accelerated by the momentum equation. This is accomplished by using a hydrostatic function, which is updated for each material in eos.m

Method 1 (particles get to accelerate and move around)

By default, particles.static = 0 so the hydro.m will include it in the list of particles accelerated pressure gradients, and hydro_Lorentz electromagnetic forces.

Method 2 (all particles for this material are static, no acceleration)
Setting a material (1 in this example) make it not move, or not accelerate, use

```
matter(1).hydrostatic = 'stationary';
```

It sets particles.static for these particles to 1.0 so they are not accelerated.

```
matter(1).hydrostatic = 'none';
```

Method 3 (this item is on a timer, and is static until the time is exceeded)
You can set item(ii).static_time to a scalar or vector (has to match number of particles in this item). This only works in flipping a particle that is otherwise dynamic into a static particle if the run time is less than the static_time set in the input file. It won't make a particle dynamic, if the material based hydrostatic model says it is a static particle.

Method 4 (this item is always static)
Set item(ii).isstatic = 1; %defaults to 0 otherwise

3.3.4.3 Viscosity

In your input file

1. `physics.viscosity=1;`

3.3.4.4 Thermal Conduction

In your input file

1. `physics.thermal_conduction = 1;`

For every material for which thermal conduction is to be modeled, (material 2 in this case), set a scalar value or choose a thermal conductivity model.

2. matter(2).thermal_conductivity = 100;

The current choices besides setting a scalar value are 'Braginskii', 'braginskii_B', and 'tabular'. Braginskii_B just means we are using the coefficients parallel, perpendicular, and cross with respect to the magnetic field. Otherwise a scalar value is computed in an unmagnetized plasma. Note that if tabular is specified, the equation of state model has to be set to tabular as well.

3.3.4.5 Ionization

In your input file

1. physics.ionization = 1;

3.3.4.6 Conductivity

In your input file

1. physics.conductivity = 1;

3.3.4.7 Friction

In your input file

1. physics.friction = 1;

3.3.4.8 Diffusion

In your input file

1. physics.diffusion = 1;

3.3.4.9 Radiation

In your input file

1. physics.radiation = 1;
2. Set the radiation model
   a. physics.radiation_model = 'optically_thin' (needs matter(ii).opacity_Planck_emis)
   b. physics.radiation_model = 'thin_or_thick' sets to optically thin until the optical depth of the radiation is less than 4h for the particle, then permanently sets it to optically thick and uses a diffusion model using Eq. 16. (needs matter(ii).opacity_Planck_emis, .opacity_Planck_abs, .opacity_Rosseland)
   c. physics.radiation_model = ‘Bremsstrahlung’
   d. To turn on multigroup radiation, physics.radiation_multigroup = 1 (only works for Bremmstrahlung at the moment)
3. matter(ii).opacity_Planck_emis = ...
   a. 'tabular' only if matter(ii).eos = 'tabular' also
   b. any numerical value that you think is appropriate
   c. 0 if you want radiation turned off for a particular material. This is the default value.
4. matter(ii).opacity_Planck_abs = ... % Also see the next section on radiation absorption
   a. 'tabular' only if matter(ii).eos = 'tabular' also
   b. any numerical value that you think is appropriate

   c. 0 if you want radiation turned off for a particular material. This is the default value.
5. matter(ii).opacity_Rosseland = ...
   a. 'tabular' only if matter(ii).eos = 'tabular' also
   b. any numerical value that you think is appropriate
   c. 0 if you want radiation turned off for a particular material. This is the default value.
6. Timing radiation to be turned on/off: The optically thin model can be put on a timer so that radiation transport is turned on and off. Set solver.timer_radiation_start = , solver.timer_radiation_stop for the times to start/stop radiation. By default these are 0 and Inf, respectively.

3.3.4.10 Radiation absorption

To set multigroup radiation absorption (from define_transport_models.m) –

1. physics.radiation_multigroup = 1
2. Set the absorption opacity
   a. matter(ii).opacity_planck_abs = 'Bremsstrahlung' %good for a plasma with no line radiation
   b. matter(ii).opacity_planck_abs =
   c. matter(ii).opacity_planck_abs='tabular' (not yet supported, but probably from propaceos table)

You need to turn on rays as well. If you see this, ask Jason to update the documentation.

3.3.4.11 Phase

In your input file

1. `physics.phase = 1;  %this turns on tracking on phase`

3.3.4.12 Ablation

In your input file

1. `physics.phase = 1;  %this turns on tracking on phase`

Then set each material to a phase or set a phase model. Currently you have 3 options:
matter(ii).phase = 'temperature' it will default to a scalar value of 2 (gas) if not specified
matter(ii).phase = 0, 1, 2 will set all particles to a scalar value picked with 0,1,2 as the choice
matter(ii).phase = 'ablation' like temperature model but phase can only stay or increase in value

Assuming you have set the material to temperature or ablation models, then you need to set
matter(ii).T_melting will default to 273.15
matter(ii).T_boiling will default to 373.15

The models work like this

```
%track phase of particles (solid, liquid, gas, plasma (0, 1, 2, 3, etc)
```

```
if physics.phase
   if strcmpi(matter(ii).phase,')
      matter(ii).phase_fun = @(particles,range) ...
         (particles.T(range) > matter(ii).T_melting) + ...
         (particles.T(range) > matter(ii).T_boiling);
    elseif strcmpi(matter(ii).phase,') %this model maintains a history of the particle phase
so that it can only stay or increase it's value
      matter(ii).phase_fun = @(particles,range) ...
         max((particles.T(range) > matter(ii).T_melting) + ...
         (particles.T(range) > matter(ii).T_boiling), (particles.phase(range)>0));
   else
      matter(ii).phase_fun = @(particles,range) matter(ii).phase;
   end
end
```

3.3.4.13 Chemistry

In your input file, for whatever material is going to conduct chemistry (2 in this example),

1. matter(2).chemistry = nonequilibrium;

There is much work to do here. Probably for Bryan Winterling

1. You then set the species.
2. The particles need to take these as properties
3. Functions need to be defined for the forward reaction rates
4. Functions need to be defined for equilibrium constants
5. They may need to be defined for mass action laws
6. These functions need to be integrated using ode45 or ode130
7. Species continuity functions need to be written.
8. These species continuity equations need to communicate with the global equation, or particle mass maybe, so that everything is conserved at the particle level.
9. See my algorithm in my physical notebook on reaction rate implementation
10. The energy equation needs to account for energy changes due to changes in species concentration, factoring in heats of formation and stuff like that. It needs to be done in a way consistent with rate of change of internal energy, not rate of change of enthalpy. See Anderson's Hypersonic text on dh/dt form of energy equation and for guidance on all equations.
11. These equations need to be assembled in a manner like the algorithm define_equations_of_motion.m This is not an easy thing to learn how to do until that algorithm is digested. Num num. Gobble gobble. Note to self. Get more sleep tonight.

3.3.4.14 Circuit

In your input file

1. physics.circuit = 1; %This tells SPFmax that you are hooking a circuit up to the model.
2. Next set the type of circuit, choices are constant, constant_voltage, rlc, charger_1 (only constant_voltage is working correctly yet)

```
circuit(1).type = 'constant_voltage'; %'rlc';
```

3. Next determine how the connection is made between the circuit and the grid. The default is just a plane of points at a minimum x, y, z or maximum x, y, z location. If so, you don't have to specify anything. Otherwise, you can 'grab' a rectangular region of points by specifying circuit(1).positive_connection_type = 'box' as below, and repeat for negative connection type. The circuit is connected at two locations, one each for the

positive and negative lead. It is recommended that if possible, orient the items (like an electrode) in such a way that you can identify an x, y, or z plane.

```
%circuit(1).positive_connection_type = 'box';
%circuit(1).negative_connection_type = 'box';
```

4. Now tell it which blocks are connected for the positive and negative leads:

```
circuit(1).positive_lead_block = 1;
circuit(1).negative_lead_block = 3;
```

5. If your connection types are the default, now tell it which side of these blocks are to be used, choices are xmin, xmax, ymin, ymax, zmin, or zmax as below. Commented out is the alternative, where you set a bounding box. You use the bounding box if you set the connection type to 'box'

```
circuit(1).positive_lead_side  = 'zmin';  %could be faces of brick (xmin, xmax, ymin, etc) or some custom thing
circuit(1).negative_lead_side  = 'zmin';

%bounding box for selecting coil particles connected to circuit
% circuit(1).pos_x0 = -4;  circuit(1).pos_dx = 8;
% circuit(1).pos_y0 = 0.1; circuit(1).pos_dy = 10;
% circuit(1).pos_z0 = -50; circuit(1).pos_dz = 6;
%
% circuit(1).neg_x0 = -4;   circuit(1).neg_dx = 8;
% circuit(1).neg_y0 =-10.1; circuit(1).neg_dy = 10;
% circuit(1).neg_z0 = -50;  circuit(1).neg_dz = 6;
```

6. Next specify the unit vector normal to the surface for the current/field flow at the boundary where the circuit is connected. this has to be done for both the positive and negative leads.

```
%unit normal for current going into/out of electrode
circuit(1).pos_nx = 0;
circuit(1).pos_ny = 0;
circuit(1).pos_nz = -1;
circuit(1).neg_nx = 0;
circuit(1).neg_ny = 0;
circuit(1).neg_nz = -1;
```

7. Next up is to provide the circuit parameters for the circuit type you specified. See below for simple examples, and the table for parameters for each model. Not that the parameters are set as follows: circuit(1).your_parameter_needed_here = your_numerical_value_here; %your comment here if you want, all in SI units

```
circuit(1).current = 1000; %amps %constant if constant model, otherwise integrated
circuit(1).voltage = 200;
```

circuit type (specified as a string)	parameters needed
constant	current
constant_voltage	voltage
rlc	resistance, inductance, capacitance
charger_1	voltage (0-100000), switch_time (1e-9 – 700e-9)

3.3.4.15 Manual current density model (without using Maxwell equation solver)

For now we just have 1 function set up. It's basically a sinusoidally decaying current pulse. Set these things:

physics.update_current_manually = 1;
circuit.frequency = ;
circuit.decayrate = ;
circuit.lengthscale = ; %needed for optional matter(ii).conductivity = 'pulsed'
circuit.area = ; %needed for optional matter(ii).conductivity = 'pulsed'
circuit.npx = ; %unit vector defines direction of current, using npx, npy, npz as the components of this vector
circuit.npy = ;
circuit.npz = ;

%see define_transport_models and define_physics_models for how these functions are set up

%defaults are set in set_item_defaults
```
%     circuit = set_default_if_needed(circuit,ii,'frequency',0);
%     circuit = set_default_if_needed(circuit,ii,'decayrate',0);
%     circuit = set_default_if_needed(circuit,ii,'lengthscale',0);
%     circuit = set_default_if_needed(circuit,ii,'area',0);
%     circuit = set_default_if_needed(circuit,ii,'npx',0);
%     circuit = set_default_if_needed(circuit,ii,'npy',0);
%     circuit = set_default_if_needed(circuit,ii,'npz',0);
%     circuit = set_default_if_needed(circuit,ii,'current',0);
%     circuit = set_default_if_needed(circuit,ii,'voltage',0);
```

Also paired with this is the ability to set the conductivity model so the voltage drop is constant.

matter(ii).conductivity = 'pulsed'; %whatever ii needs to be. i'd use it for the plasma but not the conductors. the electrodes probably should have a constant conductivity model

3.3.5 Special Discussion about Rays

Okay, this is brief for now, but if you do radiation absorption, maxwell equation solver, or fusion/fission charged particle deposition, you need to turn rays on. See below for code fragments for the input file. There are lots of ways to set up rays but below is the best method. Warning: you can easily exceed your computer's memory by making bad choices below, especially with hscale, nH, nPol, nAz, and number_of_rays

physics.rays = 1; %ray tracing

%there are various ways to create rays, and I think the best way is to snap a ray 'block' to an item and let the rays get generated based on the local geometry of the item

% see below

%snap a ray object to this item to make a bunch of rays.

rayblock(1).type = 'snap2item';

rayblock(1).item = 1; %1, 2, 3, or whatever item you are snapping this to.

rayblock(1).hscale = 3; %Scales the effective hsize of the ray points compared to particles. This is controlled by scaling hsize in the input for the ray geometry

rayblock(1).nh = 6; %sets the radius of the ray, by multiplying nh times 'h' to model how far away from particle source the radiation is modeled. If h gets smaller, so will the radius of the ray, so plan accordingly

rayblock(1).nPol = 4;

rayblock(1).nAz = rayblock(1).nPol*2; %2x as many as nPol gives even angular resolution in polar and azimuthal directions

%rayblock(1).type = 'snap2particles'; %snaps a number of rays to particles in a way that is hopefully evenly spaced.

rayblock(1).cycle_ray_positions = 0; %for snap2particles, this cycles the ray positions every hydrostep, by snapping to different particles each step or every few steps.

%rayblock(1).type = 'mesh'; %snaps a number of rays to particles in a way that is hopefully evenly spaced.

%rayblock(1).number_of_rays = 50; %approximate number of rays to distribute evenly in target and liner

rayblock(1).use_ion = 0;

rayblock(1).use_neutron = 0;

rayblock(1).use_current_density = 0;

rayblock(1).use_em_radiation = 1;

rayblock(1).raytype = '4pi';

3.3.6 Input variables

3.3.6.1 Initial conditions for particles

For material particles, you can set the initial density and temperature for each item as follows

item(1).rho = 1; %kg/m^3

item(1).T = 300; %Kelvin

and so on. Everything defaults to zero otherwise. You can also set up initial values for vectors as follows

item(1).vx0 = 5;

item(1).vy0 = 5;

item(1).vz0 = 5;

would set the initial velocity to 5 m/s for all three components.

item(1).jx0 = 25;

item(1).jy0 = 52;

item(1).jz0 = 100;

sets the initial current density for all particles as above.

3.3.6.2 Specialized functions for initial conditions

You can also set up any initial condition using a specialized function. The functions are stored in SPFMax/specialized_functions/

Here are the steps for using a specialized function for setting spatially dependent initial conditions

1. Set the initial value (e.g. item(1).rho = …) to a string which corresponds to the name of a function stored in the spfmax subdirectory 'specialized_functions'
2. for vector quantities, set the x0 value to the string, like item(4).vx0 = 'crazymanfunction' or item(27).jx0 = 'weirdalfunction';
3. Then set the custom parameters in a structured array.
    a) For density, use item(4).rho_params.<your custom parameters here>
    b) For temperature, use item(2).T_params.<your custom parameters here>
    c) For velocity, use item(3).velocity_params.<you get the idea>
    d) For current density, use item(27).currentdensity_params.<eat at joes>

e) For electric field, use item(3).Efield_params

f) For magnetic field, use item(4).Bfield params

For example

```
item(1).vx0 = 'implosion_velocity';
item(1).velocity_params.symmetry = 'cylindrical';
item(1).velocity_params.xc = 0;
item(1).velocity_params.yc = 0;
item(1).velocity_params.velocity = 1000;
```

See specialized_functions/implosion_velocity and other functions for how they are structured. It makes it very powerful for you!

For details on how the code interprets and uses your inputs, go to the file 'define_particles' and search for rho_params, T_params, velocity_params, or currendensity_params to quickly find the section that deals with these initial conditions.

Table of existing specialized functions and what they do

Specialized function	inputs	what this function does
Gresho_temperature	T_params.rho; %density T_params.g; %specific heat ratio T_params.M; %mach number T_params.MW; %molecular weight	gives temperature as a functio of radius, centered at origin, according to Gresho vortex problem, specifically the Miczek version http://www.cfd-online.com/Wiki/Gresho_vortex
Gresho_velocity	No parameters needed	pairs with Gresho_temperature, and sets a rotational velocity as a function of r.
implosion_velocity	velocity_params.symmetry; %a string, either 'cylindrical' or 'spherical' velocity_params.xc; %x center of implosion velocity_params.yc; %y center of implosion velocity_params.zc; %z center of implosion velocity_params.velocity; %implosion velocity	Makes the initial velocity vector uniform and all point towards a common axis (for cylindrical implosion) or point (spherical implosion)
ramp		

square_density		
square_wave		

3.3.6.3 Initial condition packages.m

Initial condition packages are a way to set up groups of initial conditions, and they will overwrite whatever initial conditions were otherwise set that are assigned in that group.

To use initial condition packages, settings
physics.initial_condition_package = <your string here>. Often times, other parameters have to be passed as well. Right now, the options for the string are

'spheromak_spherical'
When physics.initial_condition_package='spheromak_spherical', SPFMax sets up an initial magnetic field in the form of a spherical spheromak

Set
```
circuit.B0 = 1;  %initial magnetic field
circuit.rs = .042;  %initial target radius, i.e. spheromak flux surface
```

'magnetic_nozzle'
Sets up a magnetic nozzle which is fairly configurable.

Set
```
circuit.nozzle.length = .1;  %solenoid length
circuit.nozzle.Nturns = 20; %number of turns
circuit.nozzle.resolution = L/20/circuit.nozzle.Nturns;  %increment along the z axis for solenoid
circuit.nozzle.inlet_radius = .03;  %inlet radius of coil
circuit.nozzle.coil_current = 1e4 /2; %coil current
```

For additional details, see initial_magnetic_nozzle. You could probably set up a function handle for the radius function as a function of z, but I haven't tried that yet. The line of code in this script is around line 28, r = …. function of z

'bennet'
When physics.initial_condition_package='bennett' is set in the SPFMax input file, SPFMax assumes that the initial conditions of the plasma in the model are attempting to follow Bennett equilibrium conditions for a z-pinch. Thus, the following occurs:

a) Define_initial_condition_packages.m is triggered.
b) Takes in the Bennett input values to be used in the calculations (this includes the circuit pinch geometry and current parameters, which are passed in from the input file.
c) Computes number and line densities.
d) Computes the current density.
e) Calculates the radiation pressure.
f) Calculates the Bennett equilibrium temperature.

3.3.7 Solver variables

The 'solver' structured array is used to set parameters for numerical algorithms, set accuracy tolerances, artificial viscosity parameters, and boundary conditions.

Default values are set in set_input_defaults.m

```
if ~isfield(solver,'kernel_function'),      solver.kernel_function = 'cubicspline'; end
if ~isfield(solver,'time_integration'),      solver.time_integration = 'RKode'; end   %medium order explicit
if ~isfield(solver,'max_time_step'),       solver.max_time_step = 1e-3; end
if ~isfield(solver,'min_time_step'),       solver.min_time_step = 1e-6; end
if ~isfield(solver,'final_time'),          solver.final_time = 10e-3; end
if ~isfield(solver,'max_iterations'),      solver.max_time_steps = 1e6; end
if ~isfield(solver,'artificial_viscosity'), solver.artificial_viscosity = 'Balsara'; end
if ~isfield(solver,'artificial_heat_conduction'), solver.artificial_heat_conduction = 0; end
if ~isfield(solver,'artificial_viscosity_alpha'), solver.artificial_viscosity_alpha = 1; end
if ~isfield(solver,'artificial_viscosity_beta'), solver.artificial_viscosity_beta = 2; end
if ~isfield(solver,'artificial_heat_conduction_coefficient'), solver.artificial_heat_conduction_coefficient = 0; end
if ~isfield(solver,'Maxwell_equations'),    solver.Maxwell_equations = 0; end
if ~isfield(solver,'Material_equations'),   solver.Material_equations = 1; end
if ~isfield(solver,'max_hratio'),          solver.max_hratio = 100; end  %controls the ratio of min to max h (large particles become ballistic)
if   ~isfield(solver,'number_of_neighbors'),      solver.number_of_neighbors   =   60;   end %controls the ratio of min to max h (large particles become ballistic)
if ~isfield(solver,'dimensions'),          solver.dimensions = 3; end  %controls the number of dimensions
if ~isfield(solver,'direction'),           solver.direction = 'z'; end  %controls direction of 1D form is used
if   ~isfield(solver,'N_absorbing_layers_em'),solver.N_absorbing_layers_em   =   2;   end %number of absorbing layers for electromagnetic field boundary (far field approximation)
if ~isfield(solver,'density_smooth'),solver.density_smooth = 0; end  %smooths energy by a 1 step sph interpolation so gradients aren't discontinuous
if ~isfield(solver,'energy_smooth'),solver.energy_smooth = 0; end  %smooths energy by a 1 step sph interpolation so gradients aren't discontinuous
if ~isfield(solver,'energy_conserve'),solver.energy_conserve = 0; end  %conserve energy in kinetic and thermal
if ~isfield(solver,'momentum_conserve'),solver.momentum_conserve = 0; end  %conserve momentum
if ~isfield(solver,'dxmult'),solver.dxmult = 1; end  %multiplier for derivatives in x direction
if ~isfield(solver,'dymult'),solver.dymult = 1; end  %multiplier for derivatives in y direction
if ~isfield(solver,'dzmult'),solver.dzmult = 1; end  %multiplier for derivatives in z direction
if ~isfield(solver,'kxmult'),solver.kxmult = 1; end  %multiplier for thermal conduction in x direction
if ~isfield(solver,'kymult'),solver.kymult = 1; end  %multiplier for thermal conduction in y direction
if ~isfield(solver,'kzmult'),solver.kzmult = 1; end  %multiplier for thermal conduction in z direction
if ~isfield(solver,'jxfun'),solver.jxfun = @(t) t*0; end  %anonymous function for current density vs time, defaults to 0
if ~isfield(solver,'jyfun'),solver.jyfun = @(t) t*0; end  %anonymous function for current density vs time, defaults to 0
if ~isfield(solver,'jzfun'),solver.jzfun = @(t) t*0; end  %anonymous function for current density vs time, defaults to 0
if ~isfield(solver,'bc_periodic'),solver.bc_periodic = 0; end  %periodic boundary conditions
if ~isfield(solver,'bc_vx'),solver.bc_vx = []; end  %periodic boundary conditions
```

```
if ~isfield(solver,'bc_vy'),solver.bc_vy = []; end  %periodic boundary conditions
if ~isfield(solver,'bc_vz'),solver.bc_vz = []; end  %periodic boundary conditions
if ~isfield(solver,'bc_e'),solver.bc_e = []; end  %periodic boundary conditions
if ~isfield(solver,'bc_x'),solver.bc_x = []; end  %periodic boundary conditions
if ~isfield(solver,'bc_y'),solver.bc_y = []; end  %periodic boundary conditions
if ~isfield(solver,'bc_z'),solver.bc_z = []; end  %periodic boundary conditions
if ~isfield(solver,'bc_p'),solver.bc_p = 0; end  %periodic boundary conditions
if ~isfield(solver,'wallxmin'),solver.wallxmin = -1e30; end  %wall boundary condition for
setting static particles within an item
if ~isfield(solver,'wallxmax'),solver.wallxmax =  1e30; end  %wall boundary condition for
setting static particles within an item
if ~isfield(solver,'wallymin'),solver.wallymin = -1e30; end  %wall boundary condition for
setting static particles within an item
if ~isfield(solver,'wallymax'),solver.wallymax =  1e30; end  %wall boundary condition for
setting static particles within an item
if ~isfield(solver,'wallzmin'),solver.wallzmin = -1e30; end  %wall boundary condition for
setting static particles within an item
if ~isfield(solver,'wallzmax'),solver.wallzmax =  1e30; end  %wall boundary condition for
setting static particles within an item
```

3.3.8 Boundary conditions

Examples where needed:

1. walls (see 3.3.8.1)
2. Supersonic and hypersonic steady flow (see 3.3.8.2)
3. Rocket nozzle (see 3.3.8.3)
4. periodic boundary conditions (use ghost particles, see 3.3.8.4)
5. Taylor Green Vortex (use ghost particles, see)

3.3.8.1 Setting wall boundary conditions

What this does is freeze the motion of particles which are at or extend beyond specified boundaries. For a box, solver.wallxmin, solver.wallxmax, solver.wallymin, solver.wallymax, solver.wallzmin, solver.wallzmax set the boundaries of this box. Particles at or beyond the wall are set to static. This is handled by setting the hydrostatic_fun to

```
and(particles.x>solver.wallxmin, and( ...
   particles.x<solver.wallxmax, and( ...
   particles.y>solver.wallymin, and( ...
   particles.y<solver.wallymax, and( ...
   particles.z>solver.wallzmin,...
   particles.z<solver.wallzmax)))));
```

This is updated in eos.m

3.3.8.2 Supersonic and hypersonic steady flow

For steady, external flow problems found in atmospheric flight, the test object (missile, lifting body, wing, etc) is held stationary and the air or other gas is set to the flight velocity, much like what is done in a wind tunnel. To perform these simulations, particles need to be recycled once they pass the trailing edge of the test object. In front of the test object to be modeled, the particles need to be collimated so they provide a clean set of atmospheric conditions without

turbulence or errant expansion.

To collimate particles

```
% enable flow-freeze (collimate) for all particles x>1
collimate.enable = 1;
collimate.gt = 0;
collimate.axis = 'x';
collimate.location = 1;
```

--enable turns on/off collimation (default=off)

--gt is "greater than", meaning which "side" of the location point is frozen. gt=0 (default) means the particles at a location greater than 'location' will be frozen

--axis, the axis along which particles are frozen (default=x)

--location, where the particles become un-frozen (default = 0)

For example, this input section says "enable collimation, and freeze all particles x>1"

To recycle the particles:

```
% enable particle recycling
recycle.enable = 1;
recycle.itemlist = [3 4 5 6];
recycle.period = (length_air/4)/speed;
recycle.hh = h;
```

--enable turns on/off recycle (default=off)

--itemlist are the fluid items used to recycle. The theory is to have (at least) 4 fluid "chunks", and as the one in front gets past your area of interest, it gets gathered, and placed behind the chunk in back. All the last chunk's properties are copied to the one about to get recycled, so it's useful to have this in conjunction with collimate.

--period is how often it recycles particles. Generally this is on the order of how long it takes for 1 fluid chunk to pass through the area of interest. The code will handle any aliasing that may occur, so it just needs to be close

--hh is how far 'behind' the recycled fluid chunk is placed behind the last fluid chunk. This is usually whatever your fluid's initial h-value is.

For example, this input says "recycle particles, and place the recycled particles h behind the last set of particles; and do this every (length_air/4)/speed seconds. The fluid items are [3 4 5 6]"

3.3.8.3 Rocket nozzle

3.3.8.4 Ghost particles

If you are looking to do supersonic or hypersonic flow over a wing or body, head to the next section. Applications include periodic boundaries for ducted subsonic test cases like Couette flow, and other test cases like Taylor Green Vortex. Specifically, ghost particles are used as a reference for supplying boundary conditions for the computational domain. They are useful for inflow/outflow type boundaries, periodic boundaries where you model an infinite channel or duct, and for custom test cases like Taylor Green. Most of the time, you have to create extra ghost particle blocks using the GUI, and then while setting up the item in your input file, set item(ii).isghost = 1. This tells SPFMax that the particles in this item are ghost particles. All the other controls for ghost particles are set using the solver structured array. See below for examples of the setup.

```
%use ghost particles.   Always has to be set if ghost particles are to be used.
solver.use_ghost = 1;

%-----------------------------------------------for Taylor Green
solver.gbc = 'taylorgreen'; % string for type of boundary condition, options include periodic_x, periodic_y, periodic_z, taylorgreen
solver.g_geometry = 'convex_hull'; %or 'convexhull'; Grabs exterior hull of the particles
solver.g_hull_layers = 2; %2 or 3 are good numbers.  probably only need 2


%-----------------------------------------------for Couette
%period boundary condition parameters
solver.bc_periodic = 1; %this will cause solver.use_ghost = 1 in set_input_defaults
h=0.015;

%x0 and x1 found automatically now
% solver.bc_x0 = .0597;  %this makes it periodic in x
% solver.bc_x1 = .9552;  %an extra h is taken into account where needed using bc_h
solver.bc_h = h;
solver.gbc = 'periodic_x';
solver.g_geometry = 'sides';
solver.g_sides = 'x';  %grab the left and right sides of the blocks to use as ghost particles
solver.g_sidesN = 3;  %this gets the first 3 layers, based on solver.bc_h



solver.gbc = 'periodic_x'; % string for type of boundary condition, options include periodic_x, periodic_y, periodic_z, taylorgreen
solver.gx0 = 0; %left side of periodic boundary
solver.gx1 = 1 + h; %right side of periodic boundary
% solver.gy0 = Nan; %left  side of periodic boundary
% solver.gy1 = Nan; %right side of periodic boundary
% solver.gz0 = Nan; %left  side of periodic boundary
% solver.gz1 = Nan; %right side of periodic boundary
item(??).isghost = 1;  %otherwise defaults to 0, remember that
display('Remember to create a block  each for the left and right sides of Couette, and give them the same initial conditions as you do for the gas.');
```

```
display('You may need to create 6 blocks if your upper and lower walls are separate blocks. We''ll find out once you get it running. Just try 2 for now');
```

3.3.8.5 Virtual plenum

This is basically a model of a gas valve dumping a supply of gas over a period of time through a supersonic orifice.

3.3.9 Modification Time

Modification time means that physics, solver, and output variables can be changed on the fly. Right now it is set up to change the output time step, which is very useful for implosions. (Long output time steps during implosion, short ones at stagnation where all the action takes place on short time scales.)

modification.time(1) = <your time here>;
modification.output_time_step(1) = <your new output time step here>;

Change defaults values in set_input_defaults.m

```
%physics and output modifications
if ~exist('modification')
   modification.hey = 'how''s it going';
end
if ~isfield(modification,'time'), modification.time = inf; end %modificationification of physics or output variables at some later time. this can be a vector
if ~isfield(modification,'output_time_step'), modification.output_time_step = output.time_step; end  %modification of physics or output variables at some later time.  this can be a vector
modification.count = 1;
```

3.3.10 Default values

4. Running a Case and Post processing

4.1 Running SPFMax and run time information

4.2 The output data

5. Restarting from a time during an existing simulation

Sometimes restarting a simulation from a point just before the code crashes is useful for debugging or changing physics, and it can save a lot of wall clock time if it takes a long time to reach that point in the simulation. To do so, you load a workspace file and resume the simulation. In matlab, this is called loading a ‘checkpoint’ file and restarting. Here’s how to

do it in spfmax.

5.1 Save restart (i.e. checkpoint) files

Run a simulation, and in the input file, include the following

solver.restart_time = xxxx; %where xxxx is the first time at which restarts will be writtent

solver.restart_time_step = xxxx; %where xxxx is the time increment to write restart files

Once solver.restart_time is reached, it will save the workspace in a subfolder called ‘restart’ in your output directory for the simulation at solver.restart_time, solver.restart_time + solver.restart_time_step, and so on. The file names are prefixed with ‘restart’ and are named based on the time of the simulation, either in s, ms, μs, or ns, depending on the time scale. You can load these just like you can any matlab workspace if you want to explore the data at the command prompt. Neat, huh?

5.2 Restart a simulation

1. find the path and file name for the restart file from which you want to resume. Let’s assume this is ‘output/yoursimulation01/restart/restart020us.mat’ for sake of illustration.
2. at the command prompt, type:

spfmax(‘output/yoursimulation01/restart/restart020us.mat’,‘restart’)

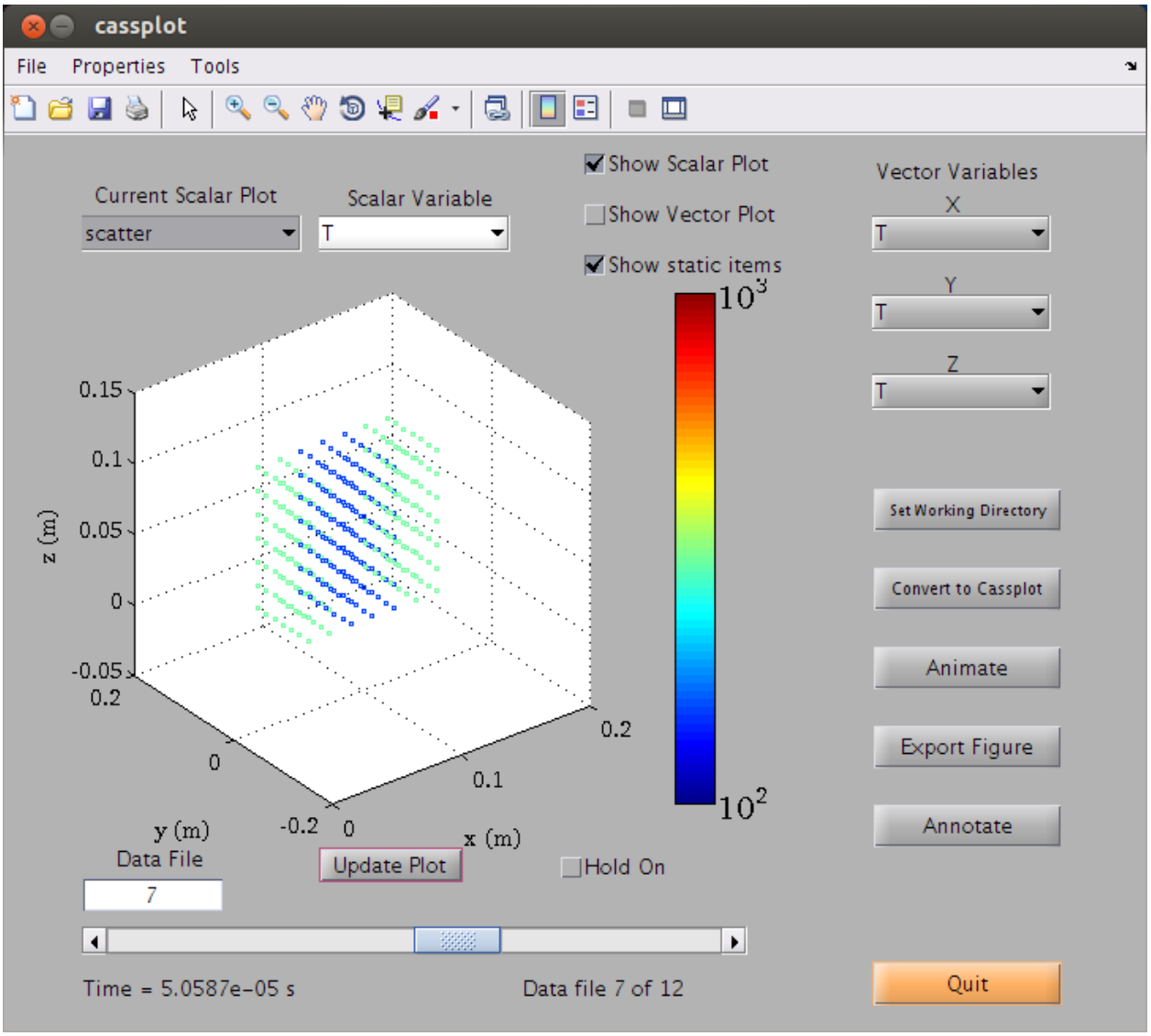
That’s it! Happy debugging!

6. Post Processing: sphplot

This is the name of the matlab package which post processes the output. It's not necessary, but makes life easier, especially if you are just learning how to analyze data.

To run sphplot, from the matlab command prompt you can either type 'sphplot(your_directory_name_here)' or just type 'sphplot'. A common way to use it is to navigate to the directory where the output data you want to study is stored, and type 'sphplot(pwd)' where pwd grabs the current directory.

The main sphplot window looks like this



Once sphplot is open, you can change working directories to look at other data by clicking the set working directory button (right middle in figure above).

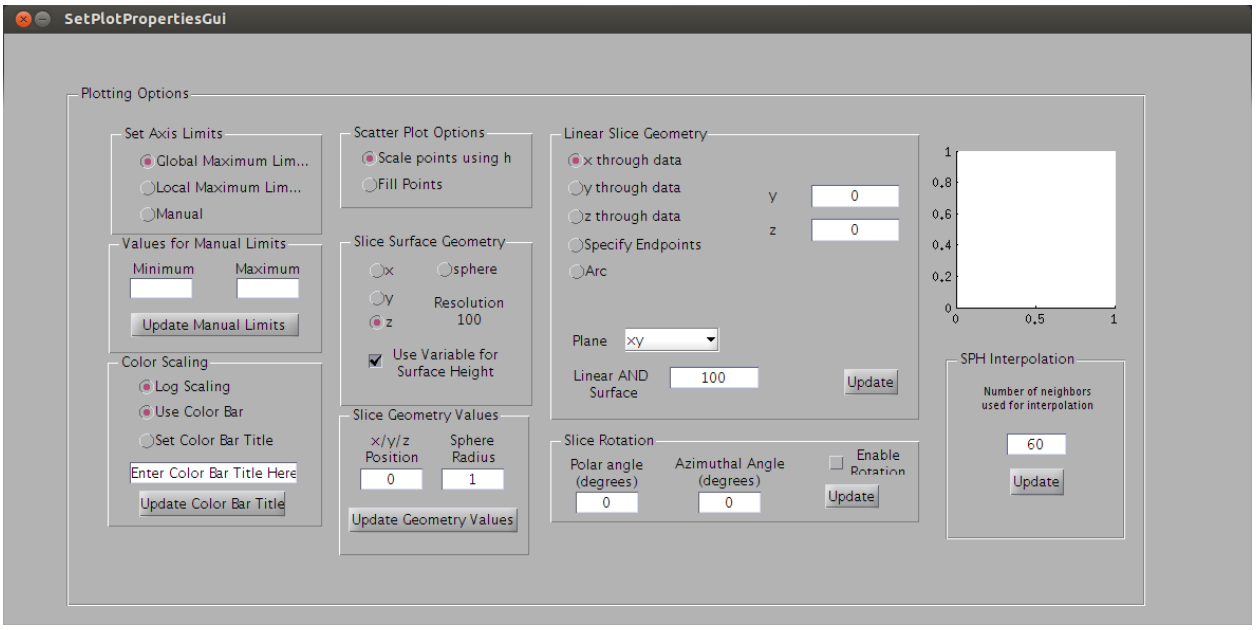
6.1 Main sphplot window

The options for choosing the type of plot are here. Your choices for the scalar plot are in the drop down menu under (current scalar plot). The variable you plot is chosen in the adjacent drop down menu, under the label 'scalar variable'. You can turn the viewing of this plot on/off by clicking the show scalar plot check box. You need to click the 'update plot' button every time you make a change.

To set the properties for a particular plot, you need to select the plot properties option from the 'properties' menu. See the next section for details.

6.2 Set Plot Properties

If you select 'set plot properties', sphplot will open the setplotpropertiesgui as shown below.



The most common features to change are on the right half of this window under linear slice geometry.

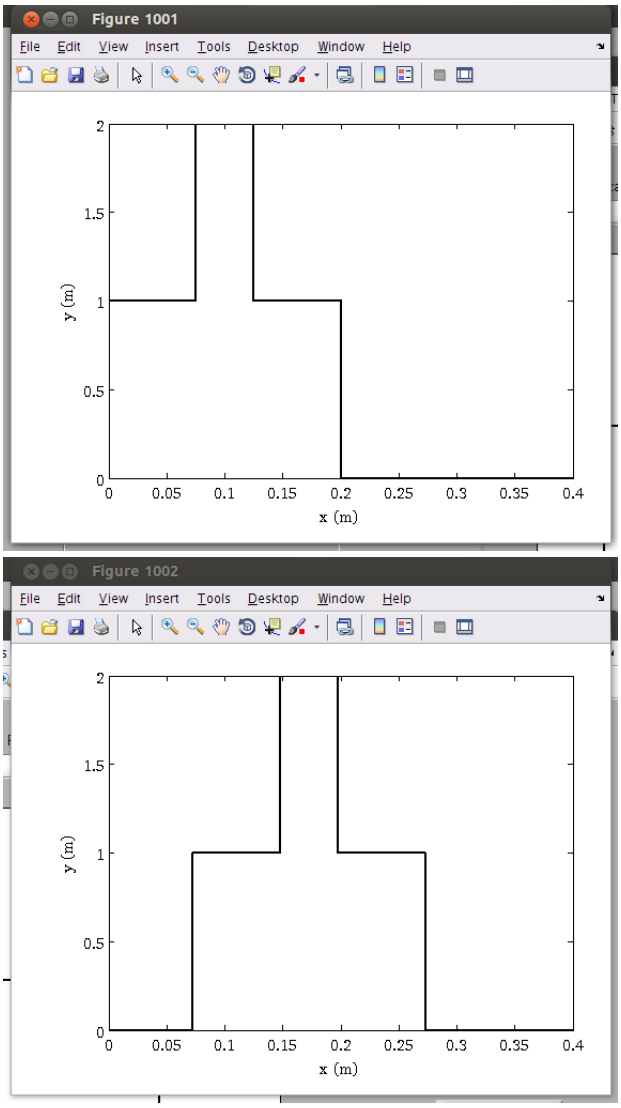
If you want to make a line slice through data (interpolate property values at evenly spaced points along a line that you define here), you set the properties of the line here.

6.2.1 Example: Post processing the square wave (advection) test case

(line slice through the data)

1. Run sphplot and open the directory containing the advection test of interest
2. Click in the slider bar to advance the time step. This just does some initialization so everything works right.
3. Click Properties -> Set Plot Properties
4. Click specify endpoints
5. Click update
6. Set x start to -.05, x end to .35
7. Set y start to .05, y end to .05
8. Set z start to .05, z end to .05
9. Set linear AND surface to 1000
10. In SPH Interpolation, set it to 1 (number of neighbors, this gives us the nearest neighbor instead of averaging across multiple neighbors)
11. Click update in BOTH the Liner Slice Geometry pane AND the SPH Interpolation pane.
12. Click back on sphplot, and select the linear slice geometry
13. Select 'ro' for the scalar variable.
14. In the 'Data File' text box, type '1' and hit enter
15. Click the 'export' button
16. In the 'Data File' text box, type '1' and hit enter

17. Click the 'export' button.
18. Now you have two figures that are independent of sphplot and can be modified/appended/polished for whatever purpose you need, such as the ones shown below. Have fun!



6.3 Annotate

6.4 Scalar plot types

6.4.1 Scatter

6.4.2 Surface

6.4.3 Slice

6.4.4 Line slice

6.5 Vector plots

6.6 Creating new Variables

7. Debugging

In order to successfully diagnose and correct problems, a working knowledge of MATLAB debugging tools is required. You need to be able to

1. make an order of magnitude estimate of the thermodynamic state and estimate of time and spatial derivatives, so you know if SPFMax is calculating them correctly
2. identify where particles are, what the properties, location, derivatives, and neighbors are.

It can be extremely tedious, but it is the only way to get the right answer for the right reason. These sections are designed to get you started. We first provide an abbreviated discussion of MATLAB, making note that there are numerous other functions and tools available. Section 2 discusses current diagnostic tools which help you figure out

7.1 Matlab commands

The `keyboard` statement can be placed in a function or script, and is used to pause that function or matlab script once MATLAB reaches that line of code. It allows you to do the following:

1. Run commands from the command prompt
2. View values of variables, arrays, and matrices by hovering the cursor over the variable

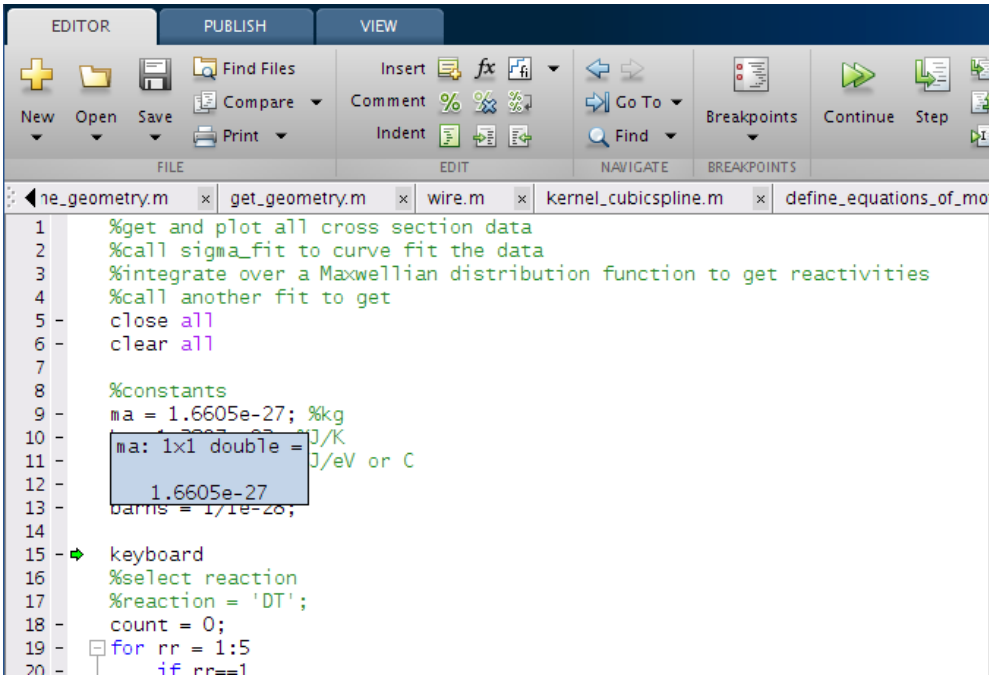


Figure 4.

7.1.1 How to tell where your keyboard statement paused the code

SPFMax uses numerous functions and scripts, and if you have put in several keyboard statements, it's easy to forget which one paused the code. The easiest way to figure out where the code is paused is to type 'dbstack'. Then you will see a short message and a hyperlink back to the script/function and line of code like this:

> In get_fusion_cross_sections_and_reactivities_Max_kappa at 15

7.1.2 Ending the debug mode initiated by 'keyboard'

Type 'dbquit' to exit debug mode.

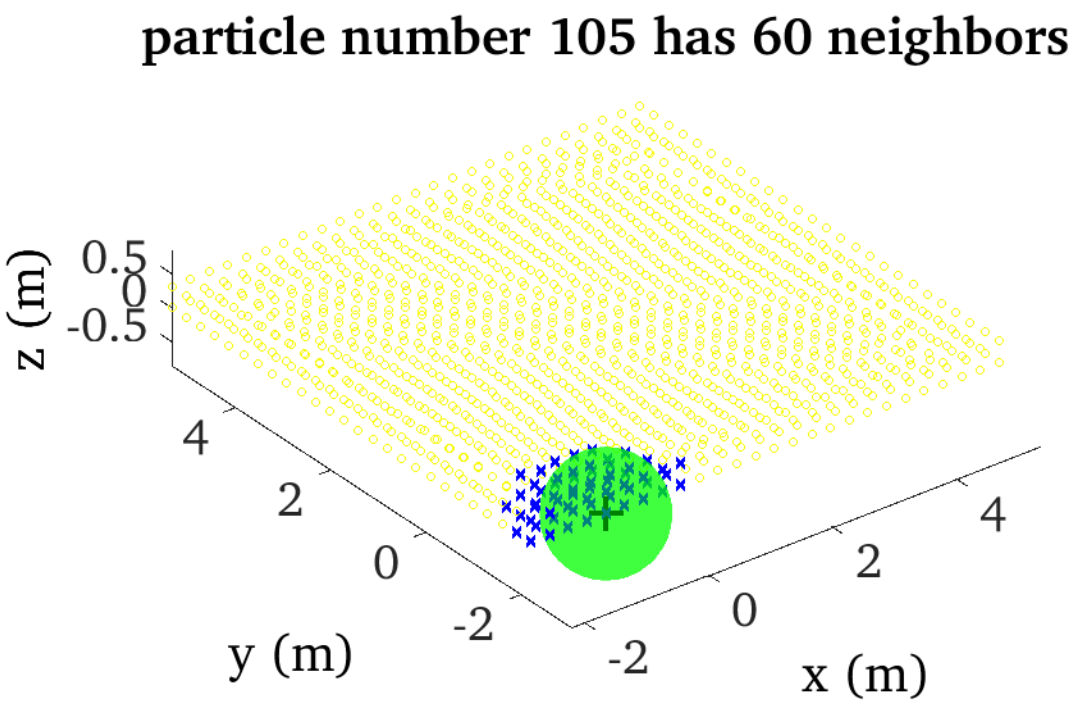
7.2 Diagnostic tools for debugging

7.2.1 plot_neighbors (in the diagnostic subdirectory)

This plots the particle (black plus sign), neighbors (blue x's), 2h compact support distance (transparent green sphere), optionally all the other particles (if the 3rd argument is used and set to anything, I like to use the number 1), and optionally the figure number for generating the plot (I like the number 300 for no good reason). In the example below, I type cd diagnostics once in keyboard mode, and type the following:

plot_neighbors(obj,105,1,300)

You use *obj* or *particles* for the 1st argument, depending on which subroutine you are in debug mode.



7.2.2 Get particle number from a spatial location

In diagnostics

get_point_nearest_location(obj,[],[4.369,4.648,0])

The 3rd argument are the x y and z coordinates that are close to where you are trying to identify the particle number.

This routine actually works for both debug mode in spfmax (using obj or particles depending on which routine you are in), or with the output files 000.mat, 001.mat, etc. If you are looking at output data, load a data file (load 001.mat, e.g.)

get_point_nearest_location(data,var,[4.369,4.648,0])

7.2.3 Get particle number from dynamic-only particle number, or vice versa

In hydro and other places, derivatives are only computed for particles that aren't walls or otherwise static. So at the beginning of the code you'll see something like range = ~logical(obj.static(1:obj.N));

Let's call this subset range of particles '*dynamic*'. Let's call the full range of particles just '*particle*'.

Often you will want to figure out why du, or some other derivative is so high or low compared to other particles, and you need to figure out exactly what particle it is in order to access all of it's properties, neighbors, etc.

To get the dynamic particle number from the Call from spfmax directory while in debug mode:

obj.obj_2_dynamic_obj(150)

This take the particle number 150 and return the corresponding dynamic particle number, id. Then du(id) would give you the acceleration of particle number 150. If you tried du(150), you would probably get the wrong acceleration.

To get the particle number from the dynamic particle number, call the function this way:

obj.obj_2_dynamic_obj([],945)

where the first argument is empty. Given dynamic particle number 945, it will return the actual particle number.

7.3 Debugging gasdynamic particle physics

Examples of problems include wall penetration, particles that are accelerated too much, etc. The way you identify the problem depends on the physics getting messed up. We will explain with examples.

7.3.1 Wall penetration

Error: wall penetrated

This is determined in hydro.m and/or boundary_condition_reflect.m

1. First, you need to set up a wall penetration test in hydro.m, perhaps after the line `dV = [du; dv; dw];`
2. to do this, pick a wall that you know gets penetrated, and determine with pencil and paper where it is, and what the logical condition would be that would establish wall penetration. say you have Rnozzlefunction(z) = sqrt(1-(z-1).^2) + 1e30*(z>1); your test would be if any(r>=R(z)), keyboard, end, where r = sqrt(obj.x(range).^2 + obj.y(range).^2); R = Rnozzlefunction(obj.z(range));
3. figure out which particle is penetrated, by stopping code at keyboard statement in hydro when the particle penetrates the wall. If you can't find where in the code is stopped, go to the command prompt and type dbstack to see which function and what line of code the keyboard statement was initiated.
4. If the code stops in a method of class particle (any function located in @particle subdirectory) the particles are called 'obj'. obj is just the name of the particle class containing all the data
5. Type this at the command prompt: find(your condition here) where your condition in the example would be r>=R(z)
6. this will give integers corresponding to the particles that penetrated the walls
7. Let's say that the offending particle is number 485

8. cd diagnostics to use some helpful tools
9. plot_neighbors(obj,485) to see the particle
10. at the command prompt, type obj.vx, obj.vy, obj.vz to see if the velocity vector is still pointing into the wall
11. we will probably at this point see that the particle velocity is pointed into the wall still, so we will need to debug boundary_condition_reflect which failed to reflect the particle. To get started, put a keyboard statement before this line `obj.vx(dfp) = vxperp + vxpara;`
12. get to the spfmax directory, then type obj.boundary_condition_reflect at the command prompt and it will stop at your new keyboard statement
13. Figure out why vperp is pointing in the wrong direction for your particle. You need to do this: plot(dfp) to figure out which index corresponds to particle 485. For example, it might be dfp(315). So vxperp(315) would give you the perpendicular component of velocity in the x direction. Is it away or into the wall? that will tell you what you need to know.

8. Adding New Physics to SPFMax

Descriptions for the process for adding new physics to SPFMax will be grouped below according to the 'family' of new physics being made. It's presented this way because the process for making changes depends on what type of change is being done. For example, families include transport models, new terms in one of the governing equations, a new circuit model, or a new EOS model. As we add new terms, they will be discussed here according to the family under which they are designated. (Just see the subsection heading that matches. If it's not there, we need to write it.)

8.1 Add a new transport model (like viscosity, thermal conduction, etc)

1. add new transport variables (if needed) to particle.m, and also add names of transport function handles, the same as in step 5.
2. write down the equations/expressions to be added to mass, momentum, and energy, where applicable
3. Add new lines to 'define equations of motion' script
4. create corresponding functions in @particle directory
5. in define_physics models, set any new physics models to the 'matter' structure, like matter(ii).viscosity_model = ...
6. In eos.m in the @particle directory, add an appropriate line to update the transport quantities every timestep.
7. In set_matter_defaults, set default values for constant transport quantities if needed

8.2 Adding in a new equation of state

1. add new parameters (if needed) to the structured array matter (if needed)
2. , and also add names of transport function handles, the same as in step 5.

3. write down the equations/expressions to be added to mass, momentum, and energy, where applicable
4. Add new lines to 'define equations of motion' script
5. create corresponding functions in @particle directory
6. in define_physics models, set any new physics models to the 'matter' structure, like matter(ii).viscosity_model = ...
7. In eos.m in the @particle directory, add an appropriate line to update the transport quantities every timestep.
8. In set_matter_defaults, set default values for constant transport quantities if needed

8.3 Adding a new circuit type

1. In define_equations_of_motion, look for `if physics.circuit` line and then the two commented out lines:

```
%    elseif strcmpi(circuit.type,'')
%        circuit.equations = @circuit_
```

2. Copy those two lines, paste them again above this, uncomment the lines, add your circuit type name between the single quotes on the first line, and circuit model ode name to the second line after circuit_
3. Copy an existing circuit model out of spfmax root diretory, paste and rename as circuit_your_awesome_model_name_here
4. Edit the new circuit model, using the guts of the old code as a guide
5. If you need to add new parameters to make it work, open set_input_defaults.m and go to the circuit parameters, then add new parameters following the syntax you see there for existing parameters.

9. Location of SPFMax Algorithms and Databases

9.1 Material Properties

9.1.1 Analytic equations of state

SPFMax/eos_xxx.m

9.1.2 Tabular equations of state

SPFMax/material_database

9.1.3 Xray Mass attenuation

SPFMax/material_database/nist_mass_attenuation

9.1.4 Fission and Fusion

SPFMax/material_database/nuclear

9.2 Governing equations and transport

9.2.1 Geometry (computational blocks)

9.2.2 Equations of motion (mass, momentum and energy)

9.2.3 Electromagnetic field (Maxwell's equations, finite difference time domain)

9.2.4 Circuit

9.2.5 Radiation (photon, ion, neutron interactions with matter)

SPFMax/define_rays – set up the geometry of the rays, ray segments

SFPMax/@ray contains the methods for determine which ray segment a particle is in, and all the stopping power models.

SPFMax/@particle/ray_setup loops over all rays, sets up the radiation/matter interaction properties (opacities for em radiation, stopping power for fast ions)

9.2.6 Fission and Fusion reactions

define_fusion_reactions sets up the reaction rates for thermal and beam target fusion models

define_fission_reactions sets up the reaction rates, scattering, and diffusion parameters for all models involving neutron transport

SPFMax/define_equations_of_motion is where the fusion and fission models are turned on as source terms

SPFMax/@particle/fusion_power.m, fusion power is computed as a source equation

SPFMax/@particle/fission_power.m, fission power is computed as a source equation

if physics.fission is turned on, then physics.neutron_diffusion is tentatively turned on as well

10. Example Input Files (The thing most people would go to first)

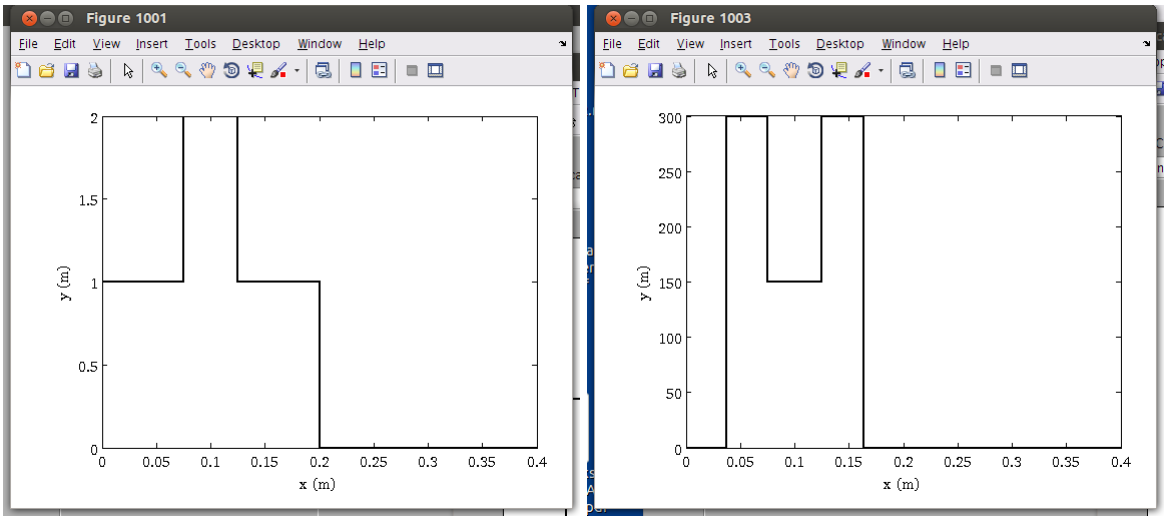
This is a list of test cases, with geometry file and input files provided so the user can reproduce the results.

10.1 Square Wave (this is an old test that needs to be redone)

The square wave test is a problem with a square wave in density initialized in a gas moving with uniform velocity bounded by walls, in which the two wall faces with unit normals parallel to the flow direction move with the flow. The temperature is initialized to be reciprocal to the density (i.e. the temperature dips when the density spikes) such that the pressure is constant. In a typical fluid solver, the sharp discontinuity in density and temperature will cause local oscillations or smearing of the wave (numerical dispersion and diffusion errors), so the sharp boundaries are not maintained as the wave propagates. Below is a screenshot of the initial density (left) and temperature (right) for the square wave problem. The initial density is either 1 or 2 kg/m^3, temperature is 300 or 150 K, and the velocity is a constant 1000 m/s. The run stops at 10^{-4} s, after the wave has propagated 1 unit length of the wave.

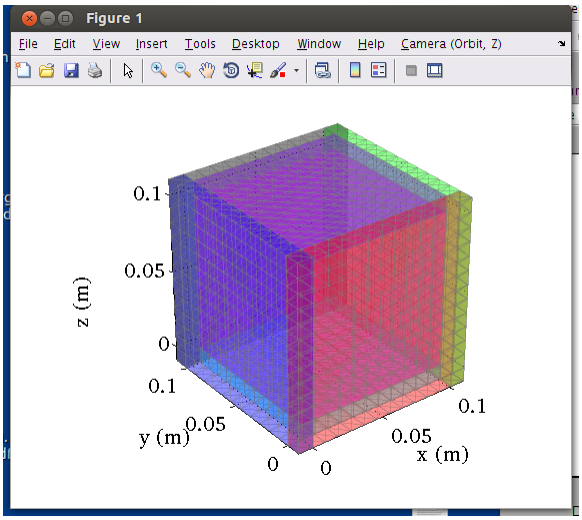
For this problem, the wave should remain a square with errors around 10^{-8}. This code was written such that advection should be accurate to floating point precision, which it appears to do. The case has been run for 343, 1728, and 10,648 particles representing the gas on a Dell Precision M6500 laptop, Dell Alienware Aurora ALX, and Dell Precision T7600 workstation. The computational times for these cases are shown in the following table

Machine	N	time (s)
M6500 laptop	343	14.9
	1728	57.1
	10648	504.2
Alienware	343	9.77
	1728	32.6
	10648	259.8
T7600 workstation	343	10.2
	1728	37.7
	10648	273.1



10.1.1 Geometry File

The geometry files are created by the script advection_test_v2, v3, or v4. To change the resolution (number of points), change the hsize relative to the scale length. We looked at 1/5, 1/10, and 1/20 of the scale length chosen. A typical set of initial blocks is as shown below.



The gas is the block in the middle, surrounded on all sides by walls consistent of two layers of points. All blocks were produced by meshgrid.

10.1.2 Input File

input files advection_08, 09, and 10 will run the square wave problem with the command such as

spfmax('example_inputs/advection_08'). The input file is as shown below.

```
%input_default.m, for prototyping spfmax
%organize input files as shown below.
%see set_input_defaults for the list of variables to assist in the
%definition

%advection8 tests square wave density

run_name = 'advection08';

%-----------------materials
```

```
%physics models, eos, transport
matter(1).name = 'argon';
matter(1).eos = 'idealgas';
matter(1).MW = 39.948;  %kj/kmol
matter(1).g = 1.3;
matter(1).chemistry = 'frozen';  %or 'nonequilibrium'
matter(1).conductivity = 1e4;  %nice plasma conductivity (gives diffusivity of 80)

matter(2).name = 'copper';
matter(2).relative_permittivity = 1;
matter(2).relative_permeability = 1;
matter(2).conductivity = 5.95e7;  %1/ohm-m
matter(2).hydrostatic = 'stationary';

matter(3).name = 'boron_nitride';
matter(3).relative_permittivity = 87; %http://accuratus.com/boron.html average of 79 and
95
matter(3).relative_permeability = 1;
matter(3).conductivity = 1e-12;  %1/ohm-m
matter(3).hydrostatic = 'stationary';

matter(4).name = 'vacuum';
matter(4).relative_permittivity = 1; %http://accuratus.com/boron.html average of 79 and
95
matter(4).relative_permeability = 1;
matter(4).conductivity = 0;  %1/ohm-m
matter(4).hydrostatic = 'stationary';


%define the item, initial conditions, state, and particle resolution

item(1).type = 'matfile';
item(1).filename = 'geometry/advection/x1wall392.mat';
item(1).matter = 'copper';
item(1).isstatic = 1;

item(2).type = 'matfile';
item(2).filename = 'geometry/advection/x2wall392.mat';
item(2).matter = 'copper';
item(2).isstatic = 1;

item(3).type = 'matfile';
item(3).filename = 'geometry/advection/y1wall392.mat';
item(3).matter = 'copper';
item(3).isstatic = 1;

item(4).type = 'matfile';
item(4).filename = 'geometry/advection/y2wall392.mat';
item(4).matter = 'copper';
item(4).isstatic = 1;

item(5).type = 'matfile';
item(5).filename = 'geometry/advection/z1wall288.mat';
item(5).matter = 'copper';
item(5).isstatic = 1;

item(6).type = 'matfile';
item(6).filename = 'geometry/advection/z2wall288.mat';
item(6).matter = 'copper';
item(6).isstatic = 1;
```

```
item(7).type = 'matfile';
item(7).filename = 'geometry/advection/gas1728.mat';
item(7).matter = 'argon';
item(7).isstatic = 0;
item(7).rho = 'square_wave';
item(7).rho_params.s1 = .025;
item(7).rho_params.s2 = .075;
item(7).rho_params.peak = 2;
item(7).rho_params.trough = 1;
item(7).rho_params.direction = 'x';
item(7).T = 11605;
item(7).T = 'square_wave';
item(7).T_params.s1 = .025;
item(7).T_params.s2 = .075;
item(7).T_params.peak = 150;
item(7).T_params.trough = 300;
item(7).T_params.direction = 'x';

%set the initial velocity
for ii = 1:7
    item(ii).vx0 = 1000;
end

%set solver variables
solver.final_time = 1e-4;

%set output variables
output.time_step = 1e-5;
```

10.2 Spherical expansion in a vacuum (ideal gas)

10.3 Spherical expansion with a tabular equation of state (Argon plasma)

10.4 Spherical Expansion from a Hemispherical Nozzle

10.5 Spherical Expansion in a Magnetic Nozzle

10.6 Steady state flow through a converging diverging nozzle

10.7 Steady state flow over an axisymmetric cone at an angle of attack

10.8 Merging of 36 plasma jets in a quasi-spherical geometry

10.9 Implosion of a z-pinch driven by an RLC circuit