Sublimation Monte Carlo (SMC)

User Manual

By

Antonio Macias

Contact Information: antonio.macias@jpl.nasa.gov | (832) 806-0343

Ocean Worlds Laboratory | NASA Jet Propulsion Laboratory | Pasadena, CA

Department of Aerospace Engineering | The University of Texas at Austin | Austin, TX

Contents

Summary 4

User Interface 4

How to compile 6

Sublimation Solver 7

Surface Data 9

  Number Flux 11

    Saturation Pressure 11

  Fraction Number 13

Monte Carlo Data 15

  Ground Boundary Condition 18

Molecular Path 20

  Common Variables 21

  Boundary Conditions 23

    Open 25

    Free 28

    Periodic 32

    JPL Experiment 39

  Intersect Surface 46

  Correct Intersection 48

Net Molecular Flux 52

  Total Mass Lost 53

  Net Mass Flux 54

  Ground Boundary Condition 55

  Sublimation Rate 55

Outputs 56

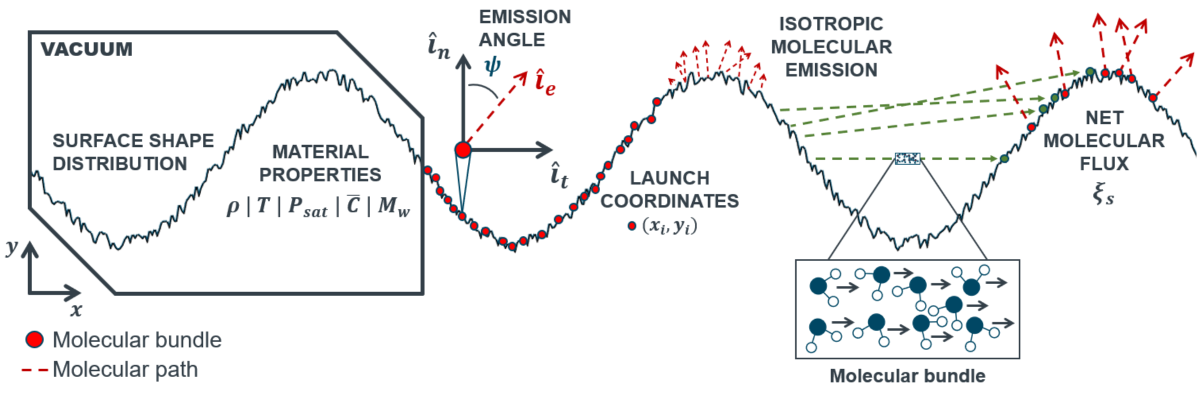
  Surface Displacement 56

  Latent Heat of Sublimation 56

Future Work 57

Summary

This document is the user manual for the Sublimation Monte Carlo (SMC) model. It contains detailed information about all the methods, including user inputs, physics, assumptions, internal algorithms, functions, boundary conditions, capabilities, compatibility, output, and future work. SMC simulates sublimation in a near vacuum background condition relevant for airless worlds such as Europa, the Moon, or Enceladus. Current material accepted is only water, but the method works for any arbitrary material given its saturation vapor pressure and molar mass. Gravity is not included in the code. SMC has running times equivalent to the state of the art of Monte Carlo methods: 600,000 simulated molecules per second while uncompiled and 34,000,000 simulated molecules per second while compiled. SMC requires the Matlab Coder ver. 5.1 or later if the code needs to be compiled and the Parallel Computing Toolbox ver. 7.3 or later if the code is running parallelized and uncompiled.



User Interface

This section presents the inputs and model structure used by SMC. This input variable is the type of simulation selected, currently, there are 4 types of simulation installed, open, free, periodic, and JPLExperiment. More information about any specific simulation in the section Boundary conditions.

```
model.type = 'periodic'; % simulation type
```

This variable creates an initial shape distribution to simulate. The struct **model.surface** contains 2 variables: **model.surface.x** and **model.surface.y** corresponding to the X and Y coordinates of the surface respectively. **Note:** the surface must meet the requirements inherent from the type of simulation selected above.

```
model.surface = generatePeniteneSurface;
```

These variables represent the time parameters of the simulation. The number of days to simulate, the number of time steps per day, and the length of a day in seconds. For Europa simulations, the time is 3.5*86400 seconds. Increasing the number of time steps per day usually increases the precision of the simulation because it simulates smaller sublimation intervals for a given surface.

```
model.time.days = 14; % number of days
model.time.daySteps = 30; % time steps per day
model.time.dayLength = 86400; % [s] length of day
```

These are the material properties, currently, the only accepted material is ice but for future work, any other material that experiences a similar sublimation behavior can be considered. When simulating SMC alone, you must input a temperature profile or a measured temperature profile. The field **model.material.initialTemperature** simulates an isothermal temperature profile at the temperature selected for the entire simulation.

```
model.material.type = 'ice'; % type of material
model.material.initialTemperature = 170 ; % [K] initial temperature
```

This is the industry standard FNUM or the numerical resolution of the code, it is the ratio of real molecules to simulated molecules. Decreasing this number increases the precision of the code and vice versa. This scale factor is linear.

```
model.resolution.FNUM = 1e17; % molecules
```

This is the ground condition, it has 2 options, on and off. The purpose is to establish a ground at y = 0 in such a way that if the surface touches the ground, it gains the ability of becoming ground. This means that a ground surface will not emit or collect molecules. If this variable is set to off, it will simulate a semi-infinite medium being infinite in the negative y direction and a limited boundary condition at the surface.

```
model.ground.zeroGround = 'on'; % ground condition
```

How to compile

SMC uses a C compiler integrated with Matlab. SMC is ready to be compiled, but there are certain conditions that need to be meet if any change to SMC is made. You can find the conditions here: https://www.mathworks.com/help/coder/ref/codegen.html.

To compile the code, insert this line on the user interface function:

```
codegen -d Modules/Temporary sublimationSolver.m -args {model}
```

There is also the choice of compiling SMC in C++ which will be slower but it removes some restrictions by adding the command **-lang:c++.** SMC is at least 2 orders of magnitude faster compiled than uncompiled. Note, that to compile the entire code, the function **sublimationSolver** must be compiled. This means that all the functions inside **sublimationSolver** will be compiled as well, and all the commands, mathematical expressions, and solutions for several variables. This method compiles in OpenMP and enables function inlining as well.

The compiler creates a new file named sublimationSolver_mex.mexw64 This file should be run instead of the singular sublimationSolver. The way to include this file into Matlab is by typing sublimationSolver_mex(model,SMC). The model struct contains the user inputs and the SMC struct contains the solver options (material properties, FNUM, time, etc.). Note that the parallel loop **will run inside the mex file** and **won't** use the Parallel Computing Toolbox from Matlab nor will activate a visualization of the number of cores that SMC is using.

Sublimation Solver

This is the main function of SMC and contains a path to all the internal functions. This function calculates the time that takes to run SMC and returns the output data from the solver. The current input call is sublimationSolver(in,SMC) where the in struct contains the X and Y coordinates, and the surface temperature or temperature profile, and the SMC struct contains the solver options (material properties, FNUM, time, etc.). The output is a struct containing the displacement of each surface element, the total mass lost in kilograms from the entire surface, the sublimation rate, and the latent heat of sublimation.

Start counting the time to run SMC:

```
fprintf('Sublimation Monte Carlo (SMC)\n'); tic
```

Discretize the surface into initial and final coordinates of each facet, calculate the area of each facet (or length for 2D), the slopes and y intercepts, and the number of molecules emitted per facet.

```
sData = surfaceData(in,SMC);
```

Assign a random position and launch angle to each simulated molecule based on an isotropic molecular emission distribution.

```
launchData = monteCarloData(sData,SMC);
```

Run the molecular path simulation in parallel for the type of simulation selected in the user interface.

```
location = parallelTracing(sData,launchData,SMC);
```

Calculate the net molecular flux and the total mass lost per facet.

```
[netFlux,in.massLost] = netMassFlux(sData,location,SMC);
```

Estimate the surface displacement.

```
in.dispFacet = surfaceDisplacement(sData,netFlux,SMC);
```

Calculate the latent heat of sublimation.

```
in.latentHeat = latentHeat(sData,nFlux,SMC);
```

Stop the clock and display the simulation time.

```
fprintf('Simulation completed in %.2f seconds\n',toc);
```

Surface Data

This function extracts the surface information based on the given shape distribution and stores it in a structure. This serves as a surface discretization. It also establishes the definitions of SMC for undefined conditions. Note that every variable here is stored in the sData structure, named after surface data, and that contains the initial and final X and Y coordinates of each facet, the area, slopes, y-intercepts, number of particles emitted per facet, angle of each facet with respect to the horizontal, and molecular mass of a water molecule.

```
xi = in.xS(1:end-1);       xf = in.xS(2:end);
zi = in.zS(1:end-1);       zf = in.zS(2:end);
sData.xi = xi;             sData.xf = xf;
sData.zi = zi;             sData.zf = zf;
```

Using the distance formula, we calculate the area/length of each facet.

```
areaFacet = ((xf-xi).^2 + (zf-zi).^2).^0.5;
sData.areaFacet = areaFacet;
```

Then using the equation of a line, we calculate the slopes and y intercept of each surface segment. The equation of a line is undefined for vertical facets, this is why SMC sets these slopes to Infinity in order to recognize them in the simulation. Even though we also set the y-intercepts to a nan value, we don't use them at all, this is just to be consistent and for the sake of completeness.

```
mF = (zf-zi)./(xf-xi);     mF(mF == -Inf) = Inf;
nF = zf - mF.*xf;          nF(isnan(nF)) = -Inf;
```

Calculate the real number flux of water molecules leaving from the surface. This variable has units of molecules per square meters per second. The number flux is calculated inside the numberFlux function which also returns the molecular mass of water molecules.

```
[nFlux,sData.mass] = numberFlux(SMC.material,in.surfTemperatures);
```

After obtaining the number flux, it must be multiplied by the area of the facet which leaves the units on molecules per second and it becomes the rate of molecules emitted per second.

```
nRate = nFlux .* areaFacet; % [molecules/s]
```

Then, based on the simulated time, the next section contains a fraction number Monte Carlo algorithm that basically leaves the units in molecules. This is the actual number of molecules to simulate per facet.

```
sData.nParticles = fractionNumber(nRate,SMC); % [molecules]
fprintf('Simulating %d molecules\n',sum(sData.nParticles));
```

Number Flux

This function calculates the number flux of water molecules leaving from each surface segment or facet. Below are some constants needed for the formulas.

```
kB = 1.380649e-23; % [J/K] Boltzman Constant

Na = 6.0221409e26; % [molecule/kmol] Avogadro's Number
```

Get the molar mass of the material, in this case snow, or ice. In this section, if you want to add another material you will need to add the molar mass.

```
if material == "ice"
    molarMass = 18.01528; % [kg/kmol]
else
    error('Select a material from the list: 1) ice')
end
Mw = molarMass/Na; % [kg/molecule]
```

Calculate the saturation vapor pressure of water above ice in Pascals as a function of temperature. This data is from Bielska 2013, but the final data will be from Murphy and Koop’s 2006 new expression.

```
satP = saturationPressure(tempProfile); % [Pa]
```

Saturation Pressure

This website contains the paper and the formula used in the lines of code shown below: https://agupubs.onlinelibrary.wiley.com/doi/full/10.1002/2013GL058474

```
% Constants
a1 = -0.212144006e2;    a2 = 0.273203819e2;    a3 = -0.610598130e1;
b1 = 0.333333333e-2;    b2 = 0.120666667e1;    b3 = 0.170333333e1;

% Reference Temperature and Pressure
refT = 273.16; % [K]
refP = 611.657; % [Pa]

% Ratio of temperature profile to the reference temperature
ratio = T./refT;

% Saturation Vapor Pressure
P = ratio.^(-1).*(a1*ratio.^(b1) + a2*ratio.^(b2) + a3*ratio.^(b3));
satP = refP*exp(P); % [Pa]
```

These formulas correspond to IAWPS Formulation from Wagner et. al. and are valid for the saturation pressure of water above ice for temperatures between 50 K and 273 K. This means that the code is validated to work for temperatures in this range. This is actually huge because also validates the code for linear surface displacement. In other words, this represents the analytic solution for a homogeneous infinite flat plate of snow sublimating in space.

Fraction Number

This function gets the real number of water molecules ejected per facet and based on the simulation time it uses a Monte Carlo algorithm by calculating random numbers and comparing them to the result of a remainder theorem. More information in the lines ahead.

This is the real number flux scaled by FNUM, which allows to simulate only a fraction of the real number of molecules.

```
rFlux = rFlux / SMC.FNUM;
```

This is a remainder theorem that acts as a Monte Carlo fraction number method in order to account for fractions of a molecule and stabilize the molecular output by adding or conserving the simulated number of molecules from each facet. We create a random number for each second to compare with the “difference” variable which should be a number between 0 and 1, similar to the random numbers.

```
roundDown = floor(rFlux);
difference = rFlux - roundDown;
rNumber = rand(round(SMC.time),length(rFlux));
```

Then, we go in time second by second, comparing the random number to the “difference” variable for all the facets and if the random number is smaller than the value of the “difference” variable for the current facet, then we add one molecule. If not, we conserve the number of molecules emitted at that second. This allows to account for molecular fractions.

```
nParticles = zeros(1,length(rFlux)); % simulated fraction number
 for ii = 1:round(SMC.time)
   tempCount = zeros(1,length(rFlux));
   for ind = 1:length(rFlux)
     % Fractions of a molecule
     if rNumber(ii,ind) <= difference(ind)
       % Added one molecule bundle
       tempCount(ind) = roundDown(ind) + 1; % [molecules]
     else
       % Kept number of molecule bundles
       tempCount(ind) = roundDown(ind); % [molecules]
     end
   end
   % Simulated number of particles per facet
   nParticles = tempCount + nParticles; % [molecules]
 end
```

The variable “nParticles” contains the final number of simulated particles emitted per facet. The units are [molecules] and these are the molecules to be tracked around the computational domain selected. The loop above is an iterative process that goes for each second for each facet and essentially finds an answer closer to the true output of the remainder theorem without the bias of a pure rounding mechanism. This way, we let the random process decide whether we simulate a greater number of molecules per second, or a lower number, based on probability distributions.

Monte Carlo Data

This function assigns a random position and launch angle to every single simulated molecule bundle. The launch angles are extracted from an isotropic distribution and the data is then stored into a column major array for faster computing.

First, we select an arbitrary random number generator seed, this command uses the time on the clock to select such seed. Assuming that the simulation time is always greater than 1 second, the seed is always going to be a new one.

```
rng shuffle
```

Then, we pre-allocate a struct called data which contains the launch information.

```
nP = sD.nParticles; % recall number of particles

data.xL = zeros(1,sum(nP)); % preallocate X coordinates

data.zL = zeros(1,sum(nP)); % preallocate Z coordinates

data.Psi = zeros(1,sum(nP)); % projected 3D launch angles

data.iL = zeros(1,sum(nP)); % launch facet index
```

This factor is just part of a formula shown ahead.

```
factor = 0; % multiplication factor used for the initial facet
```

We run through each facet.

```
for ii = 1:length(sD.xi)
```

This index oversees selecting an array of numbers to locate them into a very specific section on the column mayor array. It is totally independent on the number of molecules and number of facets. It basically selects the number of molecules from the first facet and identifies their successive location in the array. Then, for the second facet, it gets the number of molecules and identifies their location after the number of molecules in the first facet and so successively.

```
index = (factor*sum(nP(1:ii-1))+1):sum(nP(1:ii));
factor = 1; % factor used for the remaining facets
```

Computational constants: change in (x,z) coordinates of a facet.

```
% Constants
dX = sD.xf(ii)-sD.xi(ii); % delta X
dZ = sD.zf(ii)-sD.zi(ii); % delta Z
```

Select a random X coordinate along a facet.

```
data.xL(index) = sD.xi(ii) + dX.*rand(1,nP(ii)); % rng X
```

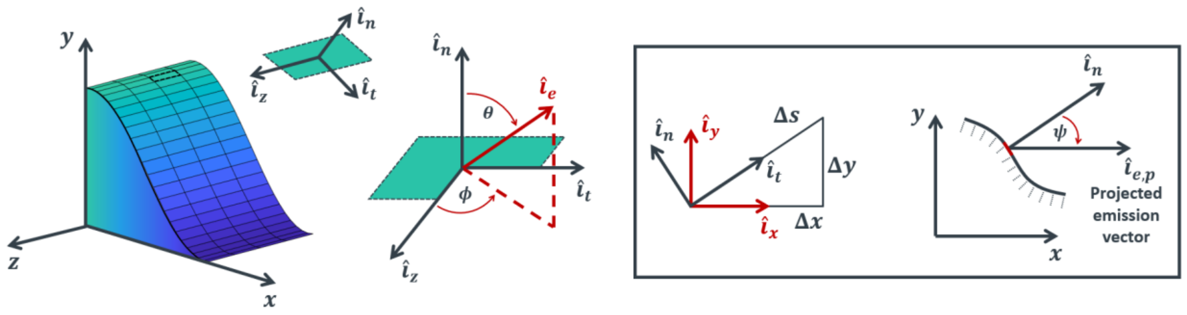
In case that the facet is vertical, the X coordinate is the same but the Z coordinate is just a random number between the initial and final Z coordinates of the facet.

```
if sD.mFacet(ii) == Inf % for undefined slopes
        data.zL(index) = sD.zi(ii) + dZ*rand(1,nP(ii));
```

Otherwise, just evaluate the X coordinate to obtain the Z coordinate using the equation of a line with the known slope and Z intercept of the line.

```
else % evaluate each X coordinate to obtain Z coordinate
        data.zL(index) = sD.mFacet(ii).*data.xL(index) + sD.nFacet(ii);
end
```

The next section is very important, this is the launch angle of each facet extracted from an isotropic distribution similar to the one shown in the Ocean Optics Webbook found in: https://www.oceanopticsbook.info/view/monte-carlo-simulation/introduction. The isotropic distribution is projected into a 2D plane in order to make the code 2D. This means that the actual scattering process is in fact in 3D but the output is only 2D because we neglect the extra coordinate. We assume that the shape is linear and infinite into and out of the page. We took a pseudo 3D approach to molecular emission by projecting 3D distribution into a 2D place before tracking the molecules.



Theta is the angle w.r.t the normal vector to the facet pointing outwards from the surface. We also pre-calculate the sine and cosine of theta to speed computations.

```
r = rand(1,nP(ii)); % randNumber for Theta
%  cos(Theta)          sin(Theta)
cT = (1-r).^0.5;       sT = r.^0.5;
```

Phi is the angle that rotates around the normal vector to the facet where $\emptyset = 2\pi r_1$ is the CDF for phi. We pre-calculate the sin of phi to speed computations.

```
Phi = 2*pi*rand(1,nP(ii)); % CDF Phi

sP = sin(Phi); % sin(Phi)
```

Finally, using the formula in these equations, we calculate the actual phi angle projected into the XZ plane.

```
% Psi: Projected 3D angle into the XZ plane

sTsP = sT.*sP; % constant

N = sTsP*(dZ) + cT*(dX); % numerator

D = sTsP*(dX) - cT*(dZ); % denominator

data.Psi(index) = atan2d(N,D); % projected 2D launch angles
```

We also record the index of the facet from which the corresponding molecule is ejected.

```
% Launch facet index

data.iL(index) = ii;
```

Ground Boundary Condition

These 2 different sections, one inside the main loop and the other outside of it, represent what we call a ground boundary condition. Basically, we have the ability of simulating a semi-infinite surface by introducing a non-morphological boundary at the bottom of the domain which corresponds to the line Z = 0. Once the surface touches the ground, it becomes ground, and then it does not release any molecules, nor collects any molecules by assuming that the ground is much cooler than the surface.

Inside the loop:

```
if ~(SMC.type == "free") || SMC.ground == "on"
    if sD.zf(ii) == 0 && sD.zi(ii) == 0
        data.xL(index) = Inf;
    end
end
```

Outside the loop:

```
if ~(SMC.type == "free") || SMC.ground == "on"
    % Particles' data that correspond to a ground segment
    ind = data.xL == Inf; % ground condition indices
    data.xL(ind)  = []; % ground X coordinates
    data.zL(ind)  = []; % ground Z coordinates
    data.Psi(ind) = []; % ground launch angle
    data.iL(ind)  = []; % ground indices
end
```

There are other sections in the code in which this ground condition is included. It refers to the previous statement in which a section of the surface considered to be ground does not displace, therefore it does not emit nor collect molecules.

Molecular Path

Particles are tracked by a process similar to raytracing in which molecules travel in a linear trajectory. According to this process a molecule cannot return to the same facet from where it was ejected. The molecularPath function is just a selector that selects the type of simulation to be simulated. The actual molecular path, tracking process, and domain construction is inside each of the 4 subfunctions.

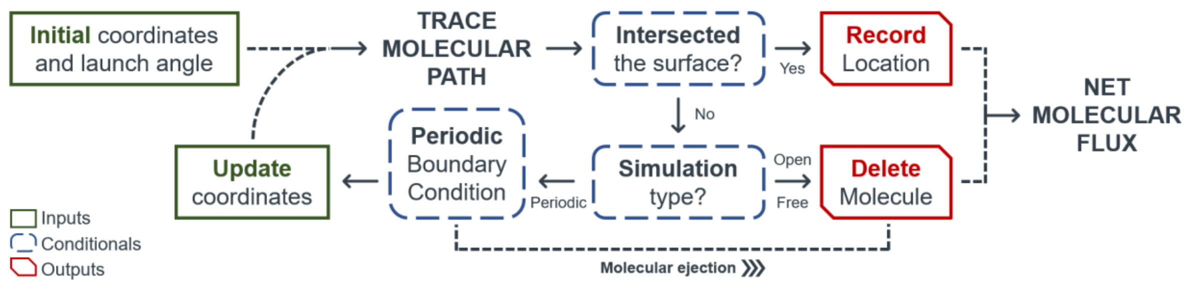
```
% Ray-trace simulation selector (boundary conditions included)
if SMC.type == "open" % real world meter-scale
        location = openTrace(sData,launchData);
elseif SMC.type == "free" % microscopic laboratory samples
        location = freeTrace(sData,launchData);
elseif SMC.type == "periodic" % planetary large-scale
        location = periodicTrace(sData,launchData);
elseif SMC.type == "JPL Experiment" % ARK chamber | Ocean Worlds Lab
        location = JPLExperiment(sData,launchData);
else
        error('Simulation type "%s" is not installed.',SMC.type)
end
```

The function openTrace contains a singular domain at the bottom similar to the zero-ground condition mentioned in the previous sections.

The function freeTrace contains no domain whatsoever. Particles after leaving the surface can only interact with other surfaces.

The function periodicTrace contains several domains. The domains on the side are periodic in which molecules can tunnel from one side to the other, the domain at the top is open, meaning that particles can just leave completely the simulation. Finally, the domain at the bottom might be closed or open depending on the user inputs.

The function JPLExperiment is a domain that resembles the Ark experiment at JPL. It behaves exactly as the openTrace simulation with the addition of a top boundary that is diffusive. After molecules arrive to this domain, they are rerouted with a new random launch angle and traced back. Molecules can return effectively to the original surface. The block diagram below shows the molecular path process for the different types of simulation. This diagram does not include the JPLExperiment simulation type.



Common Variables

This is a function that contains the common variables used by the molecular path simulation. It converts the sData struct into individual variables to improve the speed of the computation. This includes the initial and final X and Z coordinates and a couple of other conditions defined below. These are conditions to ensure that the code works always for any given surface.

This is the surface information; it contains the initial and final coordinates of the facets and both the slopes and y-intersects.

```
xi = sData.xi; xf = sData.xf; % initial|final X coordinates
zi = sData.zi; zf = sData.zf; % initial|final Z coordinates

mF = sData.mFacet; % slopes of the surface facets
nF = sData.nFacet; % z-intersects of the surface facets
```

The zero slopes condition is represented by those facets that have a zero slope. We locate those facets where the slope is zero and store them. We do this to account for floating point error in the code. Sometimes when the surface is totally horizontal, some of the molecules launched are not recognized when they interact with purely horizontal facets. This is because, the method described in the "intersectSurface" function is very sensitive to this condition. Please, see the intersect surface section to understand why this happens.

```
% Zero slopes (horizontal facets)
if any(mF == 0) % for any facet with slope equal to zero
	zM = find(mF == 0); % index of the zero-slope facets
else
	zM = 0; % variable set to 0
end
```

This condition is based on the definition of SMC to set those vertical facets to infinity since we use the equation of a line that is undefined for vertical facets. **Note**: this condition assumes that the numerical precision is so high that no particle will ever travel in either a perfect horizontal or vertical direction. Please, see the intersect surface section to understand how to apply this condition.

```
% Undefined conditions (vertical facets)
if any(mF == Inf)% for any facet with undefined conditions
        uC = find(mF == Inf); % index of the undefined facets
else
        uC = 0; % variable set to 0
end
```

Boundary Conditions

Currently, there are 4 preset boundary conditions: open, free, periodic, and JPL Experiment. The open is to run real experiments that are meter-scale in which the experimental sample is sit on top of the floor. The free boundary condition allows simulation of free-floating ice masses in space and/or microscopic ice crystals at low atmospheric pressures. The periodic boundary condition is for large-scale planetary simulations in airless worlds such as Europa and Enceladus. More information about each boundary condition can be found below. A common boundary condition for all the simulation types is the interaction between molecules and surface elements. As molecules are emitted from the surface, they can hit other surface facets and stick to them. There is a sticking coefficient usually for formulas/processes like these, but for our code, the coefficient is unity because of: https://link.springer.com/content/pdf/10.1007/BF03184673.pdf.

All the boundary conditions also share a similar structure in terms of the functions that they use, the logical process, and the variables. Using specific sections of those boundary conditions, we can create new boundary conditions. For example, if the first process in the 'new' desired boundary condition needs to check intersections between the molecule and the surface first, we will add:

```
% Direct surface intersection
[xSc,ind] = intersectSurface(m,n,xi,xf,zi,zf,mF,nF,zM,uC);

% Correct intersection as a function of the launch angle
ind = correctIntersection(xSc,ind,x1,cP(ii),iL(ii));
if ind > 0
        location(ii) = ind; % arriving location
        continue
end
```

These sections can be thought as of building blocks, hence notice how all the boundary conditions start with the same structure, and then they differ in terms of the desired logical process to follow to track the molecules. This makes it easy to install new boundary conditions such as reflective and diffusive boundaries. In a nutshell, we can just copy the entire structure of any function and then modify the logical organization inside the inner parfor loops, and we will get a new boundary condition. Just make sure that it makes sense! Note that an index of zero such as those used in conditional loops means that there is a possible intersection.

Open

Here, the experimental sample is sitting on top of a flat floor. The floor itself is a boundary condition and particles can stick to it but since it is assumed that the floor is much colder than the ice sample itself, then particles hit the floor, condense and then freeze.

Extract the common variables from the function described in the common variables section.

```
[xi,xf,zi,zf,mF,nF,zM,uC] = commonVariables(sData);
```

Extract the launch data information and store them into individual variables for speed.

```
% Launch coordinates and facet
xL = data.xL;   zL = data.zL;    iL = data.iL;
```

Calculate the sine and cosine of the launch angle upfront. This is needed for the travel direction of the molecules.

```
% Sine and cosine of the launch angle
sP = sind(data.Psi);   cP = cosd(data.Psi);
```

Then, pre-allocate the ‘location’ variable, this variable is in charge of storing the location of molecules as they are tracked across the computational domain. Specifically, it contains the index of the facet that contains the molecule. If the molecule currently simulated arrives to any facet, the location variable for the corresponding molecule will be the index of the facet.

```
location = zeros(length(xL),1); % preallocate location
```

Then, we track different molecules at the same time in random order. Molecules do not interact with each other as they are tracked across the domain. Then, they can be simulated individually at the same time using different processors, this is what this parfor loop is doing.

```
parfor ii = 1:length(xL)
```

These values are the initial launch coordinate from the particle. I extract them because inside the parfor loop, recalling the same indexed variable can turn it into a broadcast variable (similar to structs), and it will work but much slower and the result might not be the same. A broadcast variable is a variable that does not change inside a loop, so we should avoid indexing it more than once (in case that there is an array. So. this is the only place in which we index locations for variables that are going to be used at least twice inside the parfor loop.

```
% Particle's trajectory vector
x1 = xL(ii); % initial X coordinate
z1 = zL(ii); % initial Z coordinate
```

Using the concept of raytracing, we predict the travel direction of the particle by adding the X and Z components by the sine and cosine of the launch angle respectively. The distance that they travel do not matter as a line is going to be traced between the points 1 and 2.

```
x2 = x1 + cP(ii); % final X coordinate
z2 = z1 + sP(ii); % final Z coordinate
```

Using the information from points 1 and 2, we calculate the slope and z-intersect of the line that passes through those points. Note that this line will never be perfectly horizontal nor vertical due to the high numerical resolution of the variables.

```
% Line properties
m = (z1-z2)/(x1-x2); % slopes
n = z1 - m.*x1; % y-intersects
```

First, we determine ALL the intersections between the line and the surface. See the intersectSurface function for more details.

```
% Direct surface intersection
[xSc,ind] = intersectSurface(m,n,xi,xf,zi,zf,mF,nF,zM,uC);
```

Second, we find the correct intersection by using the method described in the correctIntersection function. This function also returns a zero if the particle does not interact with any surface facet.

```
% Correct intersection as a function of the launch angle
ind = correctIntersection(xSc,ind,x1,cP(ii),iL(ii));
```

Once we get the correct intersection (which corresponds to the index of the arriving facet), we store it in the location array. Then, we continue to track the next molecule since that one already sticked to that facet.

```
if ind > 0
        location(ii) = ind; % arriving location
        continue
end
```

Finally, if the particle does not arrive to any facet, we return a zero value. These zero values will be used to identify later the positions inside the 'location' array that need to be eliminated to calculate the next molecular flux. Please see the net molecular flux function for more details on this process.

```
% Particle's lost
location(ii) = 0; % lost condition
end
```

It is important to know how we define the launch angles to determine the molecular path. Basically, particles that travel 'up' have an angle between $0°$ and $180°$, and particles that travel 'down' have an angle between $0°$ and $-180°$. This is basically the output from the atan2 (for radians) and atan2d (for degrees) functions. It is the four-quadrant inverse tangent.

Free

The free boundary condition actually has no boundary conditions! The only process that matters is the molecular-surface interactions. So, we only account for molecular deposition after molecules are emitted from the facets. This method works for ice crystals as low atmospheric pressures such as snowflakes, and free-floating shapes in space. This function is exactly the same as the open function, with the difference that it does not check for molecules interactions with the floor. Since we are assuming that those molecules do not sublimate because the floor is very cold, then you will not find that the logical process to determine the surface displacement is the same for both open and free boundary conditions.

Extract the common variables from the function described in the common variables section.

```
[xi,xf,zi,zf,mF,nF,zM,uC] = commonVariables(sData);
```

Extract the launch data information and store them into individual variables for speed.

```
% Launch coordinates and facet
xL = data.xL;    zL = data.zL;    iL = data.iL;
```

Calculate the sine and cosine of the launch angle upfront. This is needed for the travel direction of the molecules.

```
% Sine and cosine of the launch angle
cP = cosd(data.Psi);    sP = sind(data.Psi);
```

Then, pre-allocate the ‘location’ variable, this variable is in charge of storing the location of molecules as they are tracked across the computational domain. Specifically, it contains the index of the facet that contains the molecule. If the molecule currently simulated arrives to any facet, the location variable for the corresponding molecule will be the index of the facet.

```
location = zeros(length(xL),1); % preallocate location
```

Then, we track different molecules at the same time in random order. Molecules do not interact with each other as they are tracked across the domain. Then, they can be simulated individually at the same time using different processors, this is what this parfor loop is doing.

```
parfor ii = 1:length(xL)
```

These values are the initial launch coordinate from the particle. I extract them because inside the parfor loop, recalling the same indexed variable can turn it into a broadcast variable (similar to structs), and it will work but much slower and the result might not be the same. A broadcast variable is a variable that does not change inside a loop, so we should avoid indexing it more than once (in case that there is an array. So. this is the only place in which we index locations for variables that are going to be used at least twice inside the parfor loop.

```
% Particle's trajectory vector
x1 = xL(ii); % initial X coordinate
z1 = zL(ii); % initial Z coordinate
```

Using the concept of raytracing, we predict the travel direction of the particle by adding the X and Z components by the sine and cosine of the launch angle respectively.

The distance that they travel do not matter as a line is going to be traced between the points 1 and 2.

```
x2 = x1 + cP(ii); % final X coordinate
z2 = z1 + sP(ii); % final Z coordinate
```

Using the information from points 1 and 2, we calculate the slope and z-intersect of the line that passes through those points. Note that this line will never be perfectly horizontal nor vertical due to the high numerical resolution of the variables.

```
% Line properties
m = (z1-z2)/(x1-x2); % slopes
n = z1 - m.*x1; % y-intersects
```

First, we determine ALL the intersections between the line and the surface. See the intersectSurface function for more details.

```
% Direct surface intersection
[xSc,ind] = intersectSurface(m,n,xi,xf,zi,zf,mF,nF,zM,uC);
```

Second, we find the correct intersection by using the method described in the correctIntersection function. This function also returns a zero if the particle does not interact with any surface facet.

```
% Correct intersection as a function of the launch angle
ind = correctIntersection(xSc,ind,x1,cP(ii),iL(ii));
```

Once we get the correct intersection (which corresponds to the index of the arriving facet), we store it in the location array. Then, we continue to track the next molecule since that one already sticked to that facet.

```
if ind > 0
    location(ii) = ind; % arriving location
    continue
end
```

Finally, if the particle does not arrive to any facet, we return a zero value. These zero values will be used to identify later the positions inside the 'location' array that need to be eliminated to calculate the next molecular flux. Please see the net molecular flux function for more details on this process.

```
% Particle's lost
location(ii) = 0; % lost condition
end
```

Periodic

This is one of the most important boundary conditions. This is used for large-scale planetary bodies by simulating an infinite surface. Particles that hit either the left or right boundaries are tunneled back into the domain and tracked again. This process can only happen twice because is a molecule hits the periodic boundary 2 times, it means that it will eventually hit the top boundary and escape. Therefore, there is no need to waste time tracking molecules over and over. There are times when a molecule will hit the periodic boundary at an angle very close to either 0° or 180° and then it will take a long time to reach the top boundary, so we made that process simpler by following a few logical steps as described throughout this section.

Extract the common variables from the function described in the common variables section.

```
[xi,xf,zi,zf,mF,nF,zM,uC] = commonVariables(sData);
```

Extract the launch data information and store them into individual variables for speed.

```
% Launch coordinates and facet
xL = data.xL;    zL = data.zL;    iL = data.iL;
```

Calculate the sine and cosine of the launch angle upfront. This is needed for the travel direction of the molecules.

```
% Sine and cosine of the launch angle
cP = cosd(data.Psi);    sP = sind(data.Psi);
```

Set the rectangular domain dimensions. By assuming that the first point of the domain is fixed at (0,0), we need the width and the height to create a rectangular domain. The height is set to be the maximum Z value of the surface. The width is to be the maximum X value of the surface or the last X data point. This creates a domain with coordinates (0,0); (w,0); (w,h); and (0,h). This represents a rectangle (technically a cube because this code is only defined in 3D but I don't want to make this complicated, the section Monte Carlo data explains the pseudo 3D approach to molecular emission in more detail. The top section of the rectangle corresponds to an open boundary in which if molecules hit it, they leave the domain. The left and right sections are the periodic boundaries. If a molecule hits either boundary it will be placed on the opposite side and tracked again. The bottom section is only active when the input variable ‘zeroGround’ is set to ‘on’.

```
h = max(sData.zi); % height of the domain
w = sData.xf(end); % width of the domain
```

Then, pre-allocate the ‘location’ variable, this variable is in charge of storing the location of molecules as they are tracked across the computational domain. Specifically, it contains the index of the facet that contains the molecule. If the molecule currently simulated arrives to any facet, the location variable for the corresponding molecule will be the index of the facet.

```
location = zeros(length(xL),1); % preallocate location
```

Then, we track different molecules at the same time in random order. Molecules do not interact with each other as they are tracked across the domain. Then, they can be simulated individually at the same time using different processors, this is what this parfor loop is doing.

```
parfor ii = 1:length(xL)
```

These values are the initial launch coordinate from the particle. I extract them because inside the parfor loop, recalling the same indexed variable can turn it into a broadcast variable (similar to structs), and it will work but much slower and the result might not be the same. A broadcast variable is a variable that does not change inside a loop, so we should avoid indexing it more than once (in case that there is an array. So. this is the only place in which we index locations for variables that are going to be used at least twice inside the parfor loop.

```
% Particle's trajectory vector
x1 = xL(ii); % initial X coordinate
z1 = zL(ii); % initial Z coordinate
```

Using the concept of raytracing, we predict the travel direction of the particle by adding the X and Z components by the sine and cosine of the launch angle respectively. The distance that they travel do not matter as a line is going to be traced between the points 1 and 2.

```
x2 = x1 + cP(ii); % final X coordinate
z2 = z1 + sP(ii); % final Z coordinate
```

Using the information from points 1 and 2, we calculate the slope and z-intersect of the line that passes through those points. Note that this line will never be perfectly horizontal nor vertical due to the high numerical resolution of the variables.

```
% Line properties
m = (z1-z2)/(x1-x2); % slopes
n = z1 - m.*x1; % z-intersects
```

First, we determine ALL the intersections between the line and the surface. See the intersectSurface function for more details.

```
% Direct surface intersection
[xSc,ind] = intersectSurface(m,n,xi,xf,zi,zf,mF,nF,zM,uC);
```

Second, we find the correct intersection by using the method described in the correctIntersection function. This function also returns a zero if the particle does not interact with any surface facet.

```
% Correct intersection as a function of the launch angle
ind = correctIntersection(xSc,ind,x1,cP(ii),iL(ii));
```

Once we get the correct intersection (which corresponds to the index of the arriving facet), we store it in the location array. Then, we continue to track the next molecule since that one already sticked to that facet.

```
if ind > 0
        location(ii) = ind; % arriving location
        continue
end
```

We check if the molecule hit the top boundary, if it does, we assign it a zero and continue. This is not necessary, but it helps to increase the speed of the code by 2 times! This is because some molecules will just directly leave through the top, so there is no need to continue running sections in this function.

We check the launch angle, if the sine is bigger than zero, it has a change of leaving, because it means the particle is traveling up! Then, we check the intersection point between the travel path and the line created from the points (0,h) and (w,h). We only need the X coordinate to check if its value is between 0 and the width of the domain, meaning that it hit the top boundary and therefore, it will be eliminated.

```
% Top Boundary Condition
if sP(ii) > 0
        xTI = ((h-z1)*(x2-x1) + (z2-z1)*x1)/(z2-z1);
        if 0 < xTI && xTI < w
                location(ii) = 0; % lost particle
                continue
        end
end
```

If the molecule did not intersect the surface initially, nor hit the top boundary directly, then it is going to hit either the left or right boundary. The cosine of the angle tells which one of the boundaries the molecule will hit. If the cosine is smaller than zero it will hit the left boundary because the molecule is traveling left, if the cosine is bigger than zero it will hit the right boundary because the molecule is traveling right. Here we apply also the periodic boundary condition. This consists on maintaining the launch angle but the launch coordinates need to be updated by the intersection point. This intersection depends on the boundary. The formulas shown in the code are explicit formulas that directly calculate the Y intersection point. The X intersection point is either zero for relocation to the left boundary or the width for relocations to the right boundary.

```
% Periodic boundary conditions
if cP(ii) < 0
        yLI = (z2*x1 - z1*x2)/(x1-x2); % Z left intersect
```

```
    x1 = w; % updated initial X coordinate
    z1 = yLI; % updated initial Z coordinate
else
    yRI = (w-x1)*(z2-z1)/(x2-x1) + z1; % Z right intersect
    x1 = 0; % updated initial X coordinate
    z1 = yRI; % updated initial Z coordinate
end
```

Since we maintain the launch angle, the slope of the line that defines the molecular path does not change. However, the Z intersect does change, so we need to update it by using the new X and Z values after applying the boundary condition.

```
n = z1 - m.*x1; % updated line's z-intersect
```

One more time we need to check first if the molecule intersected the surface after applying the boundary condition. Please see the intersectSurface function for more details.

```
% Direct surface intersection
[xSc,ind] = intersectSurface(m,n,xi,xf,zi,zf,mF,nF,zM,uC);
```

Now, at this point we do not need to use the correctIntersection function because the correct intersection in this case will be the one that has the shorter traveling distance. Please see the travelDist function for more details on this process.

```
if ind > 0
    ind = travelDist(xSc,ind,x1);
```

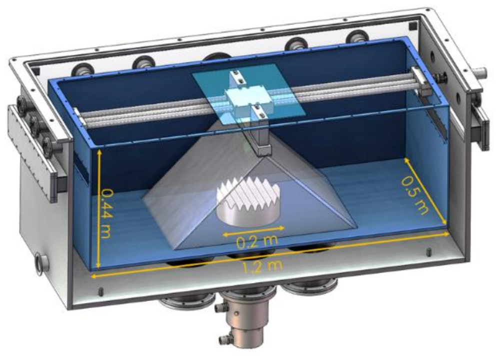
```
        location(ii) = ind; % arriving location
        continue
    end
```

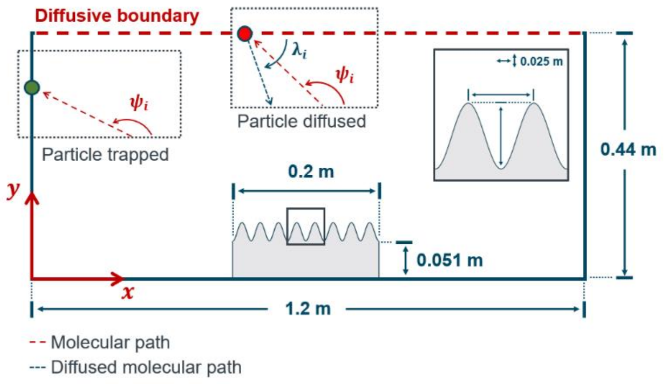
Finally, if the particle does not arrive to any facet, we return a zero value. These zero values will be used to identify later the positions inside the 'location' array that need to be eliminated to calculate the next molecular flux. Please see the net molecular flux function for more details on this process.

```
% Particle's lost
location(ii) = 0; % lost condition
end
```

JPL Experiment

This boundary condition resembles the Ark chamber at JPL. Using this boundary condition, we can: validate the code against experimental data, make predictions about results from experiments, preplan for experiments, etc. The following picture shows the Ark with dimensions and the modeling framework for this boundary condition. Here, the floor and walls act as cold traps where molecules stick to them. The top boundary is a diffusive boundary (hot) where molecules that hit it are assigned a new random launch angle and tracked again. This boundary condition is similar to the open boundary condition in terms that an ice sample is attached to the ground. However, this includes the cold traps and the diffusive top boundary, and a set of established dimensions. Surface requirements for this boundary condition are equal to the open boundary condition. We have 4 cold traps, one at each side of the snow and 2 in the walls. Note: the surface to test must be within the dimensions of the Ark and attached to the floor. Just if needed: the mean free path of water molecules in the ark is ~2 meters.





Extract the common variables from the function described in the common variables section.

```
[xi,xf,zi,zf,mF,nF,zM,uC] = commonVariables(sData);
```

Extract the launch data information and store them into individual variables for speed.

```
% Launch coordinates and facet
xL = data.xL;    zL = data.zL;    iL = data.iL;
```

Calculate the sine and cosine of the launch angle upfront. This is needed for the travel direction of the molecules.

```
% Sine and cosine of the launch angle
cP = cosd(data.Psi);    sP = sind(data.Psi);
```

Set the rectangular domain dimensions that corresponds to the Ark dimensions. By assuming that the first point of the domain is fixed at (0,0), we need the width and the height to create a rectangular domain. The height is set to 0.44 m and the width to 1.22 m. These values matter! A closer ceiling will decrease the overall sublimation rate! We create a domain with coordinates (0,0); (w,0); (w,h); and (0,h). This represents a rectangle (technically a cube because this code is only defined in 3D but I don't want to make this complicated, the section Monte Carlo data explains the pseudo 3D approach to molecular emission in more detail. The top section of the rectangle corresponds to the diffusive boundary condition, the left and right sections to the cold walls, and the bottom section to the cold floor.

```
% ARK chamber dimensions
h = 0.44; % [m] height
w = 1.22; % [m] width
```

Then, pre-allocate the 'location' variable, this variable is in charge of storing the location of molecules as they are tracked across the computational domain. Specifically, it contains the index of the facet that contains the molecule. If the molecule currently simulated arrives to any facet, the location variable for the corresponding molecule will be the index of the facet.

```
location = zeros(length(data),1); % preallocate location
```

Then, we track different molecules at the same time in random order. Molecules do not interact with each other as they are tracked across the domain. Then, they can be simulated individually at the same time using different processors, this is what this parfor loop is doing.

```
parfor ii = 1:length(xL)
```

These values are the initial launch coordinate from the particle. I extract them because inside the parfor loop, recalling the same indexed variable can turn it into a broadcast variable (similar to structs), and it will work but much slower and the result might not be the same. A broadcast variable is a variable that does not change inside a loop, so we should avoid indexing it more than once (in case that there is an array. So. this is the only place in which we index locations for variables that are going to be used at least twice inside the parfor loop.

```
% Particle's trajectory vector
x1 = xL(ii); % initial X coordinate
z1 = zL(ii); % initial Z coordinate
```

Using the concept of raytracing, we predict the travel direction of the particle by adding the X and Z components by the sine and cosine of the launch angle respectively. The distance that they travel do not matter as a line is going to be traced between the points 1 and 2.

```
x2 = x1 + cP(ii); % final X coordinate
z2 = z1 + sP(ii); % final Z coordinate
```

Using the information from points 1 and 2, we calculate the slope and z-intersect of the line that passes through those points. Note that this line will never be perfectly horizontal nor vertical due to the high numerical resolution of the variables.

```
% Line properties
m = (z1-z2)/(x1-x2); % slopes
n = z1 - m.*x1; % z-intersects
```

First, we determine ALL the intersections between the line and the surface. See the intersectSurface function for more details.

```
% Direct surface intersection
[xSc,ind] = intersectSurface(m,n,xi,xf,zi,zf,mF,nF,zM,uC);
```

Second, we find the correct intersection by using the method described in the correctIntersection function. This function also returns a zero if the particle does not interact with any surface facet.

```
% Correct intersection as a function of the launch angle
ind = correctIntersection(xSc,ind,x1,cP(ii),iL(ii));
```

Once we get the correct intersection (which corresponds to the index of the arriving facet), we store it in the location array. Then, we continue to track the next molecule since that one already sticked to that facet.

```
if ind > 0
    location(ii) = ind; % arriving location
    continue
```

```
end
```

We check if the molecule hit the top boundary, if it does, we assign it a new launch angle sampled from a uniform distribution and track it again. We check the launch angle, if the sine is bigger than zero, it has a change of leaving, because it means the particle is traveling up! Then, we check the intersection point between the travel path and the line created from the points (0,h) and (w,h). We only need the X coordinate to check if its value is between 0 and the width of the domain, meaning that it hit the top boundary and therefore, it will be diffused (assigned a new angle and tracked again).

```
% Top Boundary Condition
if sP(ii) > 0
        xTI = ((h-z1)*(x2-x1) + (z2-z1)*x1)/(z2-z1);
        if 0 < xTI && xTI < w
```

Note the negative sign in front of the angle, meaning that it will be a negative angle between 0 and -180.

```
                Psi = -180*rand; % new random launch angle
```

We maintain the arriving X and Z coordinates.

```
                x1 = xTI; % updated initial X coordinate
                z1 = h; % updated initial Z coordinate
```

However, we need to obtain the new slope and Z-intersect of the line, therefore, we obtain the new points 1 and 2 to trace the new line.

```
x2 = x1 + cosd(Psi); % updated final X coordinate
z2 = z1 + sind(Psi); % updated final Z coordinate
```

Using the new 1 and 2 points, we obtain the slope and Z-intersect of the line that defines the new molecular path and trace the molecule again.

```
% Updated line properties
m = (z1-z2)/(x1-x2); % slopes
n = z1 - m.*x1; % z-intersects
```

One more time we need to check for intersections with the surface before checking anything else. This is because other conditions might be met (such as intersections with the floor) at the same time that molecules intersect the surface. This is due to the architecture of the code of using lines to determine intersections.

```
% Direct surface intersection
[~,ind] = intersectSurface(m,n,xi,xf,zi,zf,mF,nF,zM,uC);
```

Now, at this point we do not need to use the correctIntersection function because the correct intersection in this case will be the one that has the shorter traveling distance. Please see the travelDist function for more details on this process.

```
        if ind > 0
                ind = travelDist(xSc,ind,x1);
        location(ii) = ind; % arriving location
        continue
end
```

```
end
```

Finally, if the particle does not arrive to any facet, we return a zero value. These zero values will be used to identify later the positions inside the ‘location’ array that need to be eliminated to calculate the next molecular flux. Please see the net molecular flux function for more details on this process.

```
% Particle's lost
location(ii) = 0; % lost particle
end
```

Intersect Surface

This is with no doubt one of the most important technical function in this code. It is important to understand the methods in this function to understand how this code works. The methods are simple, we only need to know that 2 lines as long as they are not parallel, will always intersect at some point in space. Therefore, we calculate all the theoretical intersects and check which of those are within the surface. This process returns all the intersections between a line and a surface. If there are intersections, we return the X coordinates of the intersection points and the corresponding index of the facets. If there are no intersections, we set return both zeros for the intersection coordinate and the index.

```
% Theoretical intersects
xI = (nF-n)./(m-mF); % X coordinates
zI = m*xI + n; % Z coordinates
```

We apply the undefined conditions from the commonVariables function here. Read both the surface data and the common variables sections to understand the reasoning behind this process. This assumes also that particles do not ever travel in perfectly vertical trajectories.

```
% Undefined conditions (vertical facets)
if uC > 0
    xI(uC) = xi(uC); % X intersect coordinates
    zI(uC) = m*xI(uC) + n; % Z intersect coordinates
end
```

Then we apply the zero slopes condition, please read the common variables section to understand the reasoning behind this process. Basically, it sets the intersection point of the facets that have a slope of zero to the initial (which is equal to the final) Z coordinate of the facet. This is done to eliminate floating point error and ensure that the code works correctly without approximating or hard coding anything.

```
% Zero slopes (horizontal facets)
if zM > 0
    zI(zM) = zi(zM); % Z coordinates (for zero-slope)
end
```

Then, we check if any of the intersections are between a rectangle created by using the initial and final coordinates of a facet. This rectangle becomes a line when the slope is zero or undefined. This is why we need the undefined conditions and the zero slopes condition.

```
% Intersections index
index = (xf-xI).*(xi-xI) <= 0 & (zf-zI).*(zi-zI) <= 0;
```

If there is any index, meaning that if there are any intersection between the line and the surface, record the X coordinate and the index of the intersection.

```
if any(index) % for existing intersections
    ind = find(index); % index of each facet intersected by the line
    xSc = xI(ind); % X coordinates of the intersection points
```

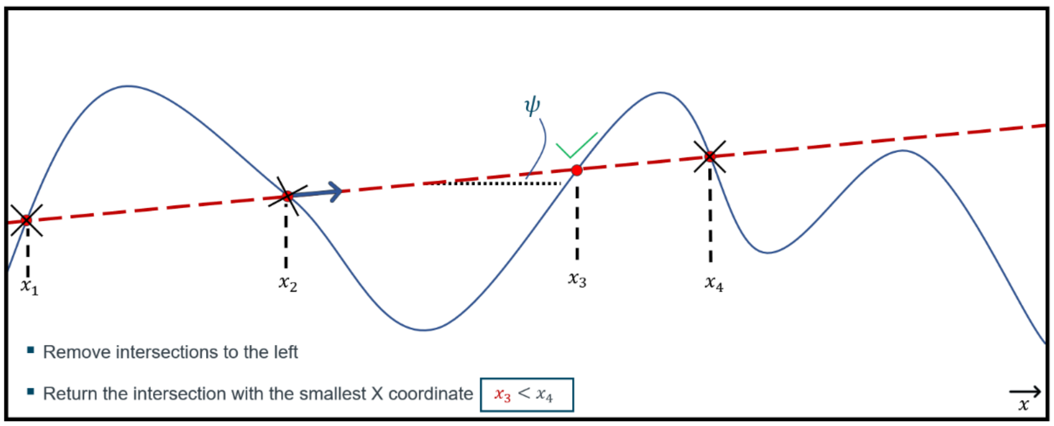
If there are no intersections, return zeros for convenience. Zeros are fast (we only need the index from these 2 but the compiler requires a value for all the outputs in order to compile the code.

```
else % for no intersections
    xSc = 0; ind = 0; % return zero index
end
```

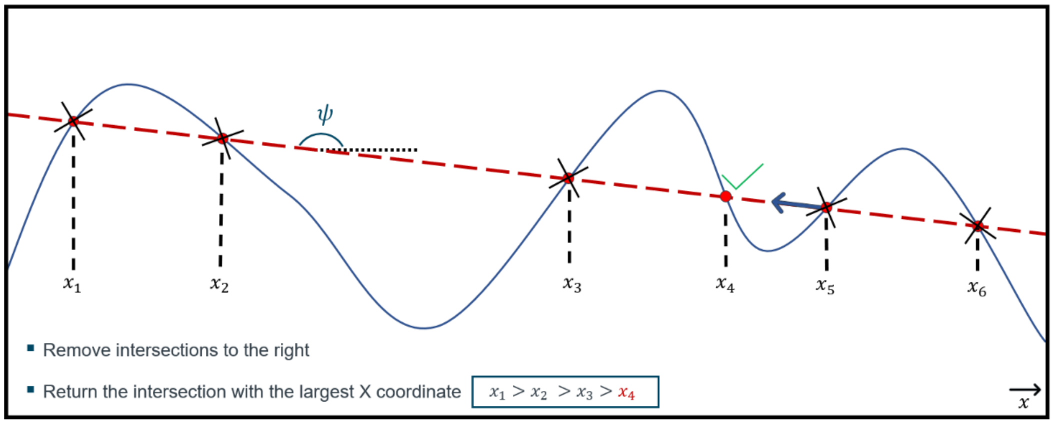
Correct Intersection

The pictures shown below illustrates the methods on this function. After knowing all the intersections, we need to determine which one is the correct. We do this by knowing the direction at which the molecule is traveling. If the absolute value of the angle is smaller than 90°, the molecule is traveling to the right, if it is bigger than 90°, the molecule is traveling to the left. From the picture above, if the molecule is traveling to the right, we discard all the intersections that are to the left, including the intersection that results from where the particle is leaving. Then, from the remaining intersections, we select the one with the smaller X coordinate. Similarly, from the bottom picture, if the particle is moving to the left, we discard the intersections that are to the right, and the one from where the particle leaves, and from the remaining, we select the one with the biggest X coordinate.

CASE 1: $|\psi| < 90°$ particle moves to the right


ψ
x_1
x_2
x_3
x_4
Remove intersections to the left
Return the intersection with the smallest X coordinate
$x_3 < x_4$
x

CASE 2: $|\psi| > 90°$ particle moves to the left


ψ
x_1
x_2
x_3
x_4
x_5
x_6
Remove intersections to the right
Return the intersection with the largest X coordinate
$x_1 > x_2 > x_3 > x_4$
x

Note: in this section LP means launch position. If there is only 1 intersection, it corresponds to the intersection from where the particle is leaving, therefore, we return a zero-index meaning that the particle didn't intersect any surface facet.

```
if length(xSc) == 1 % for LP only
```

```
    ind = 0; % return zero index
else
```

Calculate the distance between all the intersection X coordinates and the launch position.

```
% LP retrieval and storage
distX = abs(xSc-x0);
```

Then, we use it to determine which one of the X intersection coordinates corresponds to launch point and then determine its position in the array. Then, we replace the position of the index location with the first position in the array. We do this to swap the coordinates later.

```
% Index for the minimun distance
minX = min(distX);
for iD = 1:length(xSc)
    if distX(iD) == minX
        xSc(iD) = xSc(1);
        ind(iD) = ind(1);
        break
    end
end
```

Then, we replace the first position in the array with the launch position data. Basically, this next process and the previous process just swapped the initial position on the array and the position of the launch point (both the X coordinate and the index).

```
xSc(1) = x0; % remove floating point error in LP
ind(1) = iL; % recall index of the LP
```

If the molecules are moving to the left, remove those intersections and indices that are to the right, and if there are no intersections to the left, return a zero index because it means that the molecule does not intersect any surface facet.

```
if cL < 0 % for molecules moving to the left of LP
    iLeft = xSc < xSc(1); % intersections to the left of LP
    if any(iLeft) % if there are intersections to the left of LP
        % Store the intersections
        xSc = xSc(iLeft); ind = ind(iLeft);
```

Then, the correct intersection is the one that contains the maximum X coordinate! See the figure at the beginning of this section for more details.

```
        % Correct intersection contains the maximum X value
        ind = ind(xSc == max(xSc));


    else % if there are no intersections to the left of LP
    ind = 0; % return zero index
    end
```

If the molecules are moving to the right remove those intersections and indices that are to the left, and if there are no intersections to the right, return a zero index because it means that the molecule does not intersect any surface facet.

```
else % for molecules moving to the right of LP
    iRight = xSc > xSc(1); % intersections to the right of LP
    if any(iRight) % if there are intersections to the right
        % Store the intersections
        xSc = xSc(iRight); ind = ind(iRight);
```

Then, the correct intersection is the one that contains the minimum X coordinate! See the figure at the beginning of this section for more details.

```
         % Correct intersection contains the minimum X value
        ind = ind(xSc == min(xSc));

    else % if there are no intersections to the right of LP
        ind = 0; % return zero index
    end
end
end
```

Net Molecular Flux

This function uses the output from the molecular path to count how many simulated molecules arrived at each facet on the molecular path simulation. We start by determine the total mass lost from the surface, then we calculate the net mass flux from the 'location' variable as defined in the Boundary Conditions section, then we apply the ground boundary condition according to the type of simulation, and finally, we determine the sublimation rate. Specific details about these processes are in the next subsections.

Total Mass Lost

We calculate the total mass lost from the surface. To do this we find how many molecules have an index of 0, meaning that they did not intersect the surface (they escaped).

```
% Number of lost molecules
index = find(location == 0); % index of the lost molecules
nLost = length(index); % [molecules]
```

This allows to have variable size arrays that can be compiled according to the compiler requirements.

```
coder.varsize('location') % variable-size array
```

Then, we delete the simulated particles that are lost from the surface.

```
location(index) = []; % deleted lost molecules
```

Finally, we scale the particles by FNUM and multiply by the molecular mass to obtain how much mass was lost from the surface. The units are in kilograms. Note that this is not mass per surface element but total mass lost from all the facets in total.

```
% Total mass lost from the surface
massLost = sum(nLost * SMC.FNUM * sD.mass); % [kg]
```

Display the total mass lost in the command windows.

```
fprintf('  - Total mass lost: %.2e kg\n',massLost);
```

Net Mass Flux

The net mass flux is calculated by first, starting with the number of molecules launched per facet. Since these molecules are emitted from the surface, they are counted as negative mass flux, hence note the negative sign. Then, use the index from the 'location' array to determine which facet is added a molecule per index on the 'location' array. We do this for the number of particles that were left from the previous elimination process.

```
% Net mass flux
netFlux = -sD.nParticles; % molecules bundles launched per facet
for ind = 1:length(location)
    netFlux(location(ind)) = netFlux(location(ind)) + 1; % [molecules]
end
```

The result from this process is the net mass flux with units of molecules and serves to determine the sublimation rate, surface displacement, and latent heat of sublimation as described in the following sections.

Ground Boundary Condition

The ground boundary condition says that if both the coordinates (initial and final) of a facet are zero, that facet does not play a role in surface evolution. Hence, the net flux is set to zero to ensure that it does not affects future computations. Technically, these facets are not part of the surface, they are part of the ground. Meaning that if a facet touches the ground, it gains the ability of becoming ground. Note that this boundary condition is incompatible with the free simulation type.

```
% Ground boundary condition
if ~(SMC.type == "free") || SMC.ground == "on"
    netFlux(sD.zi == 0 & sD.zf == 0) = 0; % [molecules]
end
```

Sublimation Rate

The sublimation rate is the rate at which mass leaves the surface. Note that I use the average sublimation rate from across the surface, not the sublimation rate per facet. Although this can be obtained by removing the mean function. The sublimation rate can be obtained by scaling the net flux by FNUM, multiplying by the molecular mass and dividing by the simulated time.

```
% Sublimation rate
sRate = mean(netFlux * SMC.FNUM * sD.mass / SMC.time); % [kg/s]
```

Display the sublimation rate per simulated time period.

```
fprintf('  - Average sublimation rate: %.2e kg/s\n',sRate)
```

Outputs

Surface Displacement

We assume that the surface facets displace in the normal direction from a vector pointing outwards from the surface. Here we only calculate how much each facet displace. The sign means in the direction that it displaces. A negative sign means recession while a positive sign means surface growth. The formula uses the net flux scaled by FNUM multiplied by the molecular mass and then divided by the area of a facet and the material density. The units for displacement are in meters.

```
% Mass transport
massTransport = netFlux * SMC.FNUM * sData.mass; % [kg]


% Surface displacement
displacement = massTransport./sData.areaFacet/SMC.rho; % [m]
```

Latent Heat of Sublimation

As molecules strike adjacent surfaces, they increase the local energy at the impact locations. Similarly, as molecules leave a surface, they take away heat from the sublimation location. This process is described by the latent heat of sublimation. We calculate that by using the latent heat of sublimation at the triple point. The latent heat of sublimation is a function of temperature but not strongly, therefore, it is safe to assume that we can use this fix value.

```
% Latent Heat of Sublimation at T = 273.15 K.
Lv = 2.838e6; % [J/kg]
```

The latent heat of sublimation can be calculated by scaling the net flux by FNUM, multiplying it by the latent heat of sublimation at the triple point and by the molecular mass, and dividing by time.

```
% Molecular Latent Heat of Sublimation
Lh = Lv * netFlux * SMC.FNUM * sData.mass / SMC.time; % [W]
```

Future Work

The SMC code is very complete. Everything is functional as of 04/24/2021 and has been bug free since 11/15/2020. The code is also extremely optimized. SMC has running times equivalent to the state of the art of Monte Carlo methods: 600,000 simulated molecules per second while uncompiled and 34,000,000 simulated molecules per second while compiled. Things that could be implemented in the future are:

- New boundary conditions.
- Atmospheric Sublimation!
  - This is a big one but basically means writing a whole new code because the physics is different. Therefore, this is more like having a code that works for everything!
- **There is always room for innovation, email me with ideas!**