Optimization, development, and validation of the contamination transport simulation code

Brieda, Lubos, Laugharn, Marc

Lubos Brieda, Marc Laugharn, "Optimization, development, and validation of the contamination transport simulation code," Proc. SPIE 11489, Systems Contamination: Prediction, Control, and Performance 2020, 114890H (2 September 2020); doi: 10.1117/12.2566538

SPIE.

Event: SPIE Optical Engineering + Applications, 2020, Online Only

Optimization, Development, and Validation of the Contamination Transport Simulation Code

Lubos Brieda and Marc Laugharn

Particle In Cell Consulting LLC, Westlake Village, CA 91362

ABSTRACT

This paper discusses recent model improvements and performance optimization implemented in the Contamination Transport Simulation Program (CTSP). We start by discussing modification of the storage octree to utilize the space-filling Morton Z curve. We then discuss several enhancements to the input file format, such as the ability to utilize mathematical expressions and to combine several short steady-state solutions to cover a longer physical time frame. An HTML-based GUI is described. We then introduce a new molecular model based on materials-specific activation energy and the ability to use partial pressure curves to control material adhesion. Subcycling for particulate motion and a new model for particle surface adhesion are also covered. The paper finishes with a brief discussion of an ongoing experimental test campaign.

Keywords: spacecraft contamination, vacuum environment, molecular transport, particulates, scatter, high performance computing

1. INTRODUCTION

The presence and transport of molecular and particulate contaminants is a topic of concern to many industries, including ground-based nanoprocessing and spacecraft operations. Contaminants are any foreign materials that exist as molecular films, finite sized dust particulates, droplets, or in the case of cryogenic surfaces, ice. Their presence can lead to a transmission loss in optical instruments, scattering of light, off-nominal performance of thermal control surfaces, electrical shorts, limited translational rate in mechanical actuators, and possible material degradation in the case of chemically reactive species. It is not possible to completely eliminate all sources of contaminants during manufacturing, integation, and testing. Some devices are also susceptible to contamination levels below the measurement threshold of common instruments. Numerical analysis thus becomes a crucial tool in the arsenal of a systems engineer. It can help estimate the end of life contaminant loading, or the allowable exposure time to less-than-clean environments during processing. One tool that can be used for such an analysis is our CTSP (Contamination Transport Simulation Program), which has been in active development since 2015. A recent paper[1] described the state of the code as of early 2018. The objective of this report is to summarize recent modifications and improvements. We start by discussing code parallelization, and algorithm modification to improve code performance. We then discuss updates to the molecular and particulate surface adhesion logic. We also briefly touch upon a planned experimental validation.

2. CODE OPTIMIZATION

CTSP is essentially a particle tracing code. Real-world contaminant molecules or dust particulates are represented by simulation particles, each having some position and velocity $\vec{x}$ and $\vec{v}$, as well as other applicable properties such as mass or a characteristic size. The simulation essentially consists of many time steps during which new particles are injected into the computational domain, and the velocities and positions of the existing ones are advanced through a small Δt time interval according to the equations of motion, $\vec{x}' = \vec{v}$, and $\vec{v}' = (1/m)\sum \vec{F}$. We use a common explicit integration scheme known as *Leapfrog*,

$$\begin{aligned} \vec{v}^{k+0.5} &= \vec{v}^{k-0.5} + \vec{a}^k \Delta t \\ \vec{x}^{k+1} &= \vec{x}^k + \vec{v}^{k+0.5} \Delta t \end{aligned} \tag{1}$$

Corresponding author, lubos.brieda@particleincell.com

Systems Contamination: Prediction, Control, and Performance 2020, edited by Carlos E. Soares, Eve M. Wooldridge, Bruce A. Matheson, Proc. of SPIE Vol. 11489, 114890H
 · doi: 10.1117/12.2566538

Summary	12956			100.0%
FlyingMat::moveKernel(Particle&, double)	5415	170644902	317ns	41.8%
Surface::getLPIt(vec3<double> const&, vec3<double> const&, vec3<	1731	1829374746	9ns	13.36%
void std::__unguarded_linear_insert<__gnu_cxx::__normal_iterator<	1071	1620066732	6ns	8.27%
void std::__introsort_loop<__gnu_cxx::__normal_iterator<SurfaceE	877	179630663	48ns	6.77%
void std::vector<SurfaceElementData, std::allocator<SurfaceEleme	671	36455970	184ns	5.18%
SurfaceElement::containsPoint(vec3<double> const&, bool) const	542	99735545	54ns	4.18%
SurfaceElement::linePlaneIntersect(vec3<double> const&, vec3<do	491	1605043645	3ns	3.79%
FlyingMat::sample()	311	202	15.396ms	2.4%
FlyingMat::moveParticles(double)	308	2002	1.538ms	2.38%
FlyingMat::computeDv(Particle const&, double)	184	195774312	9ns	1.42%
FlyingMat::integrateXV(Particle&, double)	177	190755725	9ns	1.37%

Figure 1. Typical profiler output

where the subscript k corresponds to time. A quick run through a profiler clearly indicates that the vast majority of computational time is spent in the particle pushing step. As an example, Figure 1 shows the profiler output from a "Hello World" outgassing example utilizing a generic satellite model from GrabCad*. This test case used about 50,000 surface mesh element and tracked about 600,000 particles at each time step. We can clearly see that the `moveKernel` function performing the push of a single particle is responsible for almost half of the total run time.

While the actual evaluation of Equation 1 is trivial, during a single time step, each particle advances from position $\vec{x}^k$ to $\vec{x}^{k+1}$. In order to capture the presence of surface geometry, it is necessary to test this trajectory for a surface impact. In our implementation, this test involves performing *line-triangle intersection* checks to determine whether the linear segment traced by the particle intersects any surface elements. Given that a typical simulation contains several hundred thousand particles and a similar number of surface elements, it is imperative for this check to be as rapid as possible.

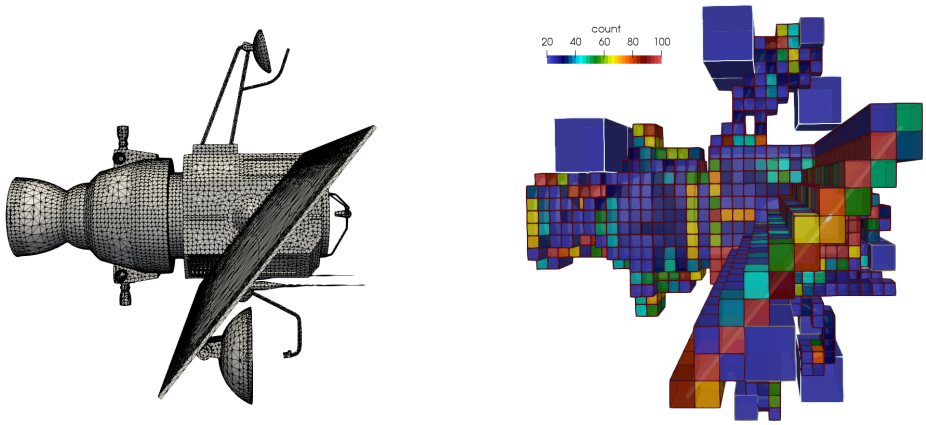
2.1 Octree

The first step is to reduce the subset of elements to check. The motion from $\vec{x}^k$ to $\vec{x}^{k+1}$ spans a bounding box and only surface elements that are, at least partially, located inside this box are candidates for intersection. A prior work[2] used a Cartesian mesh to discretize the computational volume. Each volume cell (or a node-centered control volume) held a list of surface elements located inside of it, if any. The bounding box of the particle motion was converted to a range of logical $i-j-k$ cell indexes using the volume mesh, and the particle trajectory was compared against surface elements stored by these cells.

This approach is applicable to particle-based gas kinetic codes in which the cell size is set based on local plasma density or collisional distance. However, in the predominantly free-molecular flow analysis, as encountered in contamination transport, there is no longer any characteristic length that could be used to specify a volume mesh size. For this reason, CTSP was designed from the onset as a *volume mesh free* code. A volume mesh can be specified, but is used solely, in collisionless simulations, to sample macroscopic flow properties such as density, stream velocity, and temperature for results visualization. CTSP uses an *octree* to store the surface mesh elements. An octree is a container in which the enclosing volume is divided into two equal-sized partitions in each spatial dimension, leading to eight blocks (octants) in three dimensions. Each octant can be recursively subdivided further using this same process until the number of surface elements per block is smaller than some threshold. The size of the block is related to the *level* of the particular refinement. Figure 2 shows the resulting blocks for our test geometry. The coloring indicates the number of surface elements (triangles or quadrangles) stored in each block. We can notice the presence of several large blocks containing a small number of elements. Near the satellite body, the blocks become smaller due to an increase in local density of surface elements.

Early versions of CTSP used a recursive memory data structure to store the octree. Each block was labeled either as a "leaf" or a "branch". A leaf node contained an array (implemented using a C++11 *vector* container) of data located within the octant. A programming concept known as *templates* was used to support storing arbitrary data types, since besides surface geometry, CTSP uses octrees to also store vector data needed to evaluate forces acting on particulates. A branch node, on the other hand, stored memory pointers for up to eight

*`https://grabcad.com/library/satellite-11`



(a) surface mesh (b) corresponding octree

Figure 2. Octree representation of a test satellite geometry

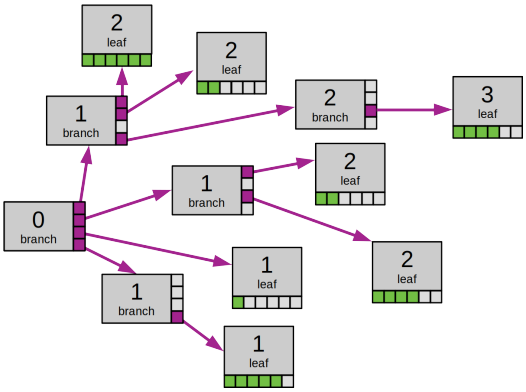


Figure 3. Illustration of a recursive octree (visualized as a quadtree) container based on memory links.

child nodes, as visualized in Figure 3. In this plot, the purple and green "slots" represent either pointers to child octants (purple) or actual data elements stored in leaf nodes (green).

This implementation directly captures the recursive structure of an octree. It is however sub-optimal for several reasons. First, the actual octants are scattered through memory. Performance of modern CPUs is often limited by a finite memory bandwidth and for this reason, CPUs utilize on-board cache to fetch data the processor is expecting to need next. With a random memory access, this prefetch fails. Performance of algorithms can often be increased ten fold or more by simply re-arranging memory access pattern to reduce these *cache misses*. A linked-list data structure, as implemented here, is not optimal, despite being relatively easy to code up. This linked list approach is also not conducive to a computation on graphics cards (GPUs) which are even more demanding of data access arrangement. CTSP does not yet support GPU processing, but this enhancement is planned for a near term future work. Finally, as can be seen from the figure, accessing data for a particular octant requires recursive traversal through the entire pointer-linked list starting with the top-level root node.

The octree algorithm was thus redesigned to store the blocks in a consecutive-memory array. Furthermore, instead of storing both the "branch" and "leaf" nodes, only the leaves are now retained. The blocks are sorted according to a well-known indexing called the Morton Z-order curve. This approach is not unique to CTSP, and is often utilized in adaptive mesh refinement codes.[3] A Morton curve is type of a *space filling* curve where connecting spatial locations according to their z value, completely fills in the volume without the curve self-intersecting. The z-curve is visualized for a two-dimensional domain in Figure 4.

The Z index of a particular point can be computed by first mapping the physical position to a logical (i, j, k) coordinate using a virtual Cartesian grid spanning the domain. Since for a uniform mesh, $x = x_0 + i\Delta x$, this

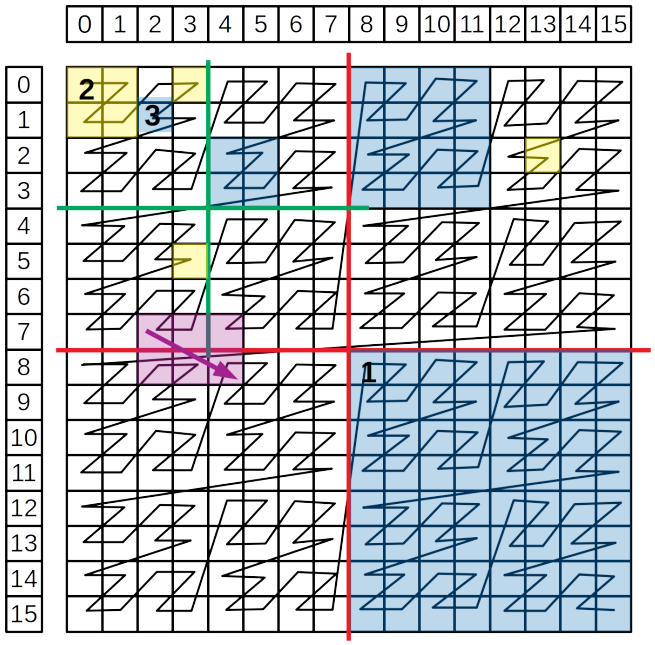


Figure 4. Illustration of the Morton Z-curve in two dimensions

computation is trivial

$$i = \left\lfloor \frac{x - x_0}{\Delta x} \right\rfloor \tag{2}$$

with similar expressions for the j and k index for the y and z direction. Here $\lfloor \cdot \rfloor = \text{int}(\cdot)$ is the floor operation that discards the fractional part for positive values. Δx is the size of the smallest possible octree block. As will be apparent shortly, the octree cannot be subdivided indefinitely, at least not without implementing custom mathematical operators. The block size can be computed from the *level* of the finest octree. At level $L = 1$, there are $2^L = 2$ blocks in each dimension. Similarly, for $L = 7$, we have $2^7 = 128$ blocks, or cells, in each direction. Block size is then obtained per

$$\Delta x = \frac{x_m - x_0}{2^L} \tag{3}$$

The Z number is computed by *interleaving* (or *mux*-ing) the bits of the three integer i, j, and k indexes.[4] Given a binary representation

$$\begin{aligned} \text{bin}(i) &= i_n i_{n-1} \dots i_2 i_1 i_0 \\ \text{bin}(j) &= j_n j_{n-1} \dots j_2 j_1 j_0 \\ \text{bin}(k) &= k_n k_{n-1} \dots k_2 k_1 k_0 \end{aligned}$$

the Z number is

$$Z(i,j,k) = k_n j_n i_n \dots k_2 j_2 i_2 k_1 j_1 i_1 k_0 j_0 i_0 \tag{4}$$

Connecting cells in order of their Z value gives us the characteristic curve shown in Figure 4.

The number of bits needed to store the Z number is $3n$. On modern PC architectures, an integer and a "long" integer are both 32-bits wide, thus limiting the (i, j, k) indexes to having at most 10 significant bits. The number of bits is also directly equal to the maximum refinement level. To illustrate this, a coarse tree with only a single subdivision in x will have only two possible i states, $i = 0$ or $i = 1$. These two possibilities can be encoded in the i_0 bit. A level $L = 2$ tree will have 4 subdivision in x with indexes 0, 1, 2, and 3. Again, using their binary representation, these values are encoded with the i_1 and i_0 bits.

To illustrate the Z number in practice, consider the two-dimensional representation in Figure 4. We start by creating blocks over the entire computational domain using some initial coarse level. Using initial level 1, we end up with four blocks, each covering a region separated by the red lines. We then iteratively insert data into the appropriate blocks. Whenever the number of items within the octant exceeds some threshold, and we have not yet reached the maximum level, we split the octant into eight new $L+1$ blocks. After data from the original block is distributed to these new containers, the original block is deleted. The new blocks are inserted into the array starting at the position of the original larger block. Due to the nature of the Z-curve, we can be assured that there will not be any overlap between the blocks. This process repeats until all data is inserted. Any empty blocks are deleted at the conclusion of the build phase. This leaves us with a possible arrangements as shown in the Figure. Here, the alternating yellow and blue shaded regions are supposed to indicate the remaining octants containing data (i.e. the leaf nodes from the prior recursive scheme). The 1, 2, and 3 labels indicate the block level.

In our C++ implementation, we take advantage of the standard library binary search algorithm to limit the range of blocks to check for insertion based on the bounding box of the to-be inserted item. For each block in this range, we utilize a `inBox` function to evaluate whether the data item is inside the block or not. For vector data, this function simply compares the point position to the octant extents, but for triangular elements, we need to implement additional logic to check for face and edge intersections.

During particle motion, we utilize a `getData(double3 xmin, double3 xmax)` function to retrieve an array of all surface elements from blocks located within the given limits. For particle push, these two coordinates correspond to the $\vec{x}^k$ and $\vec{x}^{k+1}$ position. This function works by converting the two coordinates to their Z number and then using C++ standard library `binary_search` algorithm to find the *iterator* for the starting and ending position in the ordered array of blocks. We iterate over all blocks in this range, and if the octant's bounding box intersects the desired box, all items owned by the octant are copied to a return vector. Duplicate entries, corresponding to surface elements spanning multiple octants, are deleted using C++ standard library functions `sort`, `unique`, and `erase`. The bounding box check is necessary, since in some cases, the $\vec{x}_{min}$ and $\vec{x}_{max}$ bounds may be located near-by in physical space, but far in the Z domain. This situation is encountered when a low-level boundary is crossed. The purple arrow in Figure 4 illustrates the distance covered by a single particle. While the starting and the ending points are relatively close by, the Z-curve connecting them traverses through the entire Level 1 block at $i = 8$, $j = 0$. Clearly, elements from the larger blue and the smaller yellow non-empty blocks within this quadrant should not be included in the list of elements to check.

As part of our optimization strategy, we have attempted to implement an alternate scheme not utilizing the linear iterator in order to avoid these situations. However, timing studies indicated that the attempted speedup actually resulted in an inferior performance. The likely explanation is that edge cases as discussed above are infrequent. By implementing a specialized algorithm that was perhaps more efficient for these particular cases, we increased the computational overhead for the vast majority of lookups in which near-by points are also located at adjacent Z coordinates. One popular strategy for mitigating this issue is to utilize a "tree of trees".[3] Here a coarse Cartesian grid is used to first discretize the domain. Then a unique octree is defined for each Cartesian cell. Implementing, and timing, this alternate approach remains as future work. Yet, even this intermediary version of the octree algorithm, we can already notice drastic improvement in run time. The run time for a 20,000 time step simulation of our example satellite configuration reduced from 8.2 minutes with version 0.29 (May 2019) to just over 1 minute with the latest version (1.6).

2.2 Intersection Sort

When checking for surface impact, it is not sufficient to just determine whether a surface was hit. Typical spacecraft geometries contain thin bodies, and a particle can enter, and leave, the object in a single time step. As such, it is imperative to locate the *first* surface element to be contacted. Rewriting the particle position using a parametric form,

$$\vec{x}^{k+t} = \vec{x}^k + t\vec{v}^{k+0.5}\Delta t \qquad t \in [0,1] \tag{5}$$

we need to find the impact location that minimizes the parameter t. Intersection between a line segment and a triangle can be performed using two steps. First, we determine whether the segment intersects the *plane* of the

triangle. This trivial computation is performed by substituting the equation for the line segment, Equation 5 for point $\vec{x}$ in the equation of a plane

$$\hat{n} \cdot (\vec{x} - \vec{x}_0) = 0 \tag{6}$$

and solving for the parametric t. Here $\vec{x}_0$ is a point known to be on the plane, and $\hat{n}$ is the plane normal vector. The segment intersects the plane if $t \in [0, 1]$. Next, we need to determine whether the intersection point lies within the bounds of the surface element. CTSP treats quadrangles as two virtual triangles for the sake of surface impact checking since quadrangles generated by typical FEM meshing applications tend to have a slight amount of warp. There are many standard approaches at our disposal for determining whether a point $\vec{x}_p$ lies within a triangle. For instance, we could compare the total area of the three sub-triangles formed using two vertices and the point $\vec{x}_p$. For the point to be internal, we require $A_{12p} + A_{23p} + A_{31p} = A_{123}$, where the area of a triangle is obtained per

$$A = \frac{1}{2} \Big| (\vec{x}_2 - \vec{x}_1) \times (\vec{x}_3 - \vec{x}_1) \Big| \tag{7}$$

We can also compare the bisection angles at each vertex, $\angle_{1,2p} + \angle_{1,3p} = \angle_{1,23}$ where $\angle_{i,lm}$ is approximated by the dot product $(\vec{x}_l - \vec{x}_i) \cdot (\vec{x}_m - \vec{x}_i)$. Another option includes comparing the orientation of normal vectors formed at each vertex. For an internal point, $[(\vec{x}_2 - \vec{x}_1) \times (\vec{x}_p - \vec{x}_1)] \parallel [(\vec{x}_p - \vec{x}_1) \times (\vec{x}_3 - \vec{x}_1)]$ with similar expressions existing on the other vertices.

CTSP uses the second, angle bisection, method. Yet regardless of the chosen scheme, we can clearly see that the mathematical complexity of classifying intersection point location is much greater than encountered in testing for line-plane intersection. Prior versions of CTSP combined these two algorithms into a single function that returned the parametric $t \in [0, 1]$ of an intersection point, or $t = -1$ if no intersection was found. While some optimization was implemented in the form of only checking locations for points closer than the currently-known closest intersection, $t < t_{min}$, the algorithm still resulted in an excessive number of "isPointInTriangle" checks. In this updated code version, the intersection algorithm was split into two separate loops. First, the line-plane intersection t is computed for all surface elements returned from the octree. The elements are then sorted according to the t value. Location check is then performed in order, and the search terminates once the first valid intersection is encountered.

2.3 Parallel Processing

The computational time can be further decreased by employing parallel processing. There are three technologies at our disposal: multithreading, distributed computing, and the use of graphics cards (GPUs). All modern processors (CPUs) contain multiple computational cores. Essentially, each CPU contains multiple internal processors capable of performing independent instructions. These cores can be utilized using a technique called *multithreading*. CPUs found in standard laptops contain somewhere between 3 to 8 cores, while some workstations feature a dual CPU design with around 20 cores per CPU. Still, even using these high-end processors, the total number of CPU cores is quite limited. *Distributed computing* allows us to overcome this limit by running the simulation on multiple physical computers interconnected over a network connection. Finally, graphics card feature massive number (several thousand) of relatively slow cores. They are optimized for vector calculations in which the same operation can be applied in parallel to a large set of unique, independent, data.

Recently, support for both multithreading and distributed computing has been added to CTSP. GPU parallelization is still pending. Each method has its benefits and drawbacks. With mulithreading, it is necessary to identify the algorithms that can benefit the most from a rewrite, and then modifying them to utilize multiple CPU cores. In our case, this algorithm is the code performing particle push. Even outside the free-molecular flow regime, particles do not interact with each other during the push, which involves integrating the equations of motion, Equation 1, and checking for surface impact. Therefore, instead of a single serial loop,

```
for (size_t p=0;p<num_parts;p++)
  pushParticle(particles[p]);
```

we can use the C++11 `thread` object to launch, in parallel, several copies of a `pushParticles` function

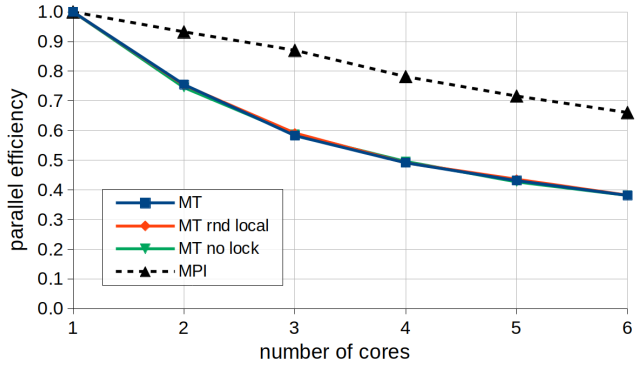


Figure 5. Parallel efficiency obtained on a 6-core i7 CPU with multithreading and MPI

```
for (size_t tid=0;tid<num_threads;tid++)
std::thread(pushParticles,tid);
```

with each thread pushing only a subset of the entire particle population,

```
size_t np_per_thread = num_parts/num_threads;
for (size_t p=tid*np_per_thread;p<min(np,(tid+1)*np_per_thread;p++)
  pushParticle(particles[p]);
```

This is essentially the approach implemented in CTSP. During its motion, the particle can undergo one or more surface impacts. Condensing particles are removed from the simulation, and the corresponding surface deposition thickness is incremented. Conversely, particles not sticking bounce off in a new random direction following the cosine-law distribution about the surface normal. Both of these operations introduce challenges to the multithreading algorithm. Due to being spawns of the same main thread, the threads have access to the same *shared* memory space. This reduces the need to duplicate input data for each individual thread but can also lead to erroneous results if *race condition* is encountered. This term describes the situation in which multiple threads attempt to *write* to the same memory location at the same time, leading to indeterminate results. Such a scenario is encountered on each particle surface impact, since particles handled by different threads can stick to the same surface element, and thus both threads need to increment that element's surface deposition counter. There are two possible workarounds. The first option is to let each thread have a local copy of the surface molecular data. The second option is to utilize a *mutex* object to serialize this critical code block. Both approaches introduce some computational overhead. The first approach requires more memory and also code to distribute, and subsequently combine, the global data. Locking a mutex also requires CPU cycles. Furthermore, the waiting threads also remain idle while the first thread is processing. In order to determine whether implementing the local-copy algorithm is beneficial, we ran the satellite simulations with the mutex-locking code commented out. This comparison can be seen in Figure 5. We can notice that there is no meaningful difference between the solid black and solid green lines. It is possible that simulations in which the ratio of particles to surface elements is greatly increased, the difference becomes more noticeable.

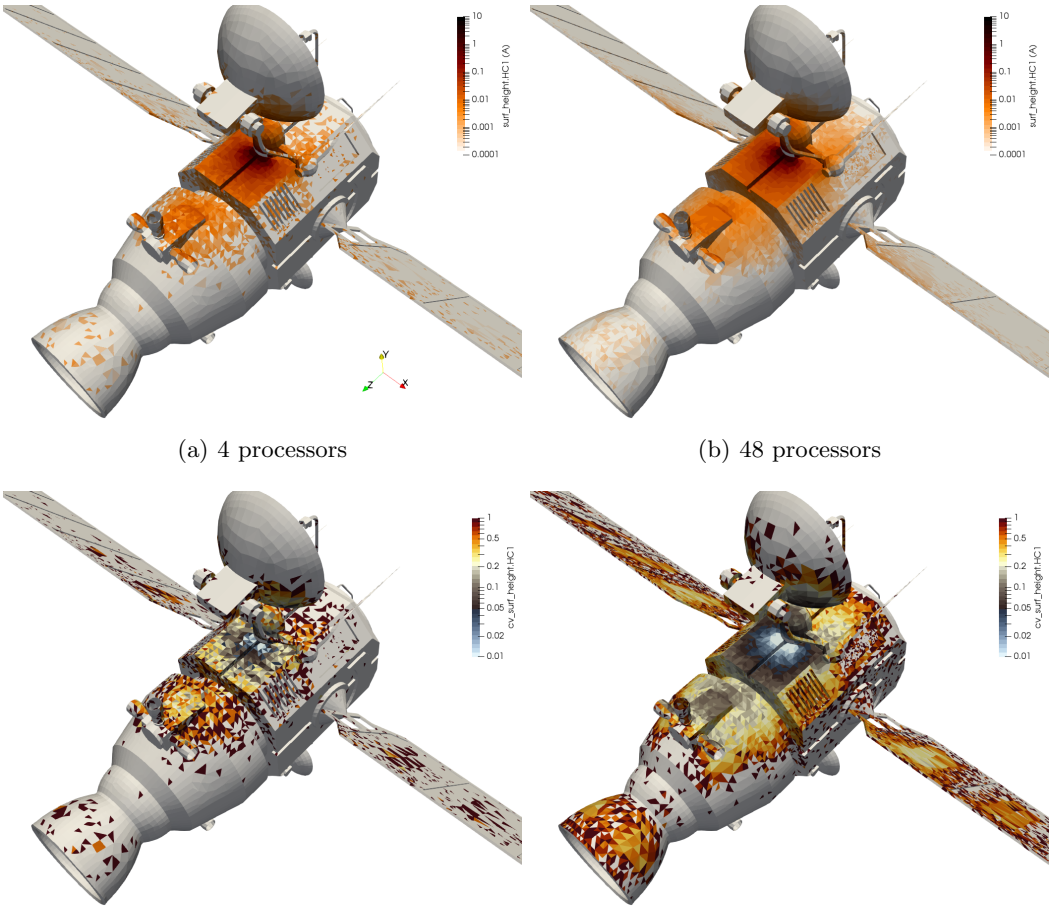
In addition, it is important to be aware of hidden serialization in standard library functions. The primary culprit is the random number generator. In our stochastic, Monte-Carlo approach, random numbers are used to determine whether a particle sticks, and to obtain the post impact velocity magnitude and direction. These numbers are produced by sampling a *random number generator*. Instead of producing truly random values, RNGs are mathematical algorithms that sample consecutive values from a particularly large sequence. After each number is generated, the sequence index needs to be incremented. This increment would lead to a race condition given a multi-thread concurrent access, and thus is internally serialized. Sharing the same generator among multiple threads can often lead to a poor parallel efficiency. However, again for this particular example, we do not see much difference between a simulation in which all threads use the same RNG object (black line)

and one with each thread having its own generator (red line). However, given the minimal impact on the code base, the code has been refactored to give each thread its own generator.

The trend shown in Figure 5 was measured on a Dell Precision workstation featuring a six-core Intel i7 8700HQ CPU. We can see that parallel efficiency, computed per

$$\eta_{par} = \frac{t_{serial}}{n_p t_p} \tag{8}$$

where t_p is the run time with n_p processors, is only 37% using all 6 cores. This behavior is expected, since parallelization has been applied to only the single algorithm responsible for pushing particles. As discussed previously, this function is responsible for under 50% of total run time. The rest of the code, which include functions that also scale with the number of particles such as mass sources or algorithms for sampling volumetric quantities, continues to run serially. Implementing multithreading in every function is not practical.



(a) 4 processors

(b) 48 processors

(c) c.v., 4 processors

(d) c.v., 48 processors

Figure 6. Deposition thickness and the coefficient of variation from a 4 and a 48-CPU run

For this reason, CTSP also supports parallel processing using the Message Passing Interface (MPI).[5] MPI is the standard communication protocol used on supercomputers, but can be used locally on a single system. Due to the free-molecular nature of contamination transport, there is no interaction between individual simulation particles outside of collisions. An increased number of particles simply reduces the statistical noise. Therefore, we can use MPI to essentially run multiple *serial* simulations, and, at completion, average the individual results. This approach, known as *ensemble averaging*, eliminates the overhead associated with MPI data transfer. As an example, Figures 6(a) and 6(b) compare surface deposition computed using 4 and 48 processors. While the

results are similar, we can notice a much smoother variation in the predicted surface contamination thickness with more processors.

Since each processor runs and entire serial simulation, we can also use the individual results to compute the coefficient of variation (normalized standard deviation),

$$c_v = \frac{\sqrt{\frac{1}{n}\sum_s^{n_s}(x_s - \bar{x})^2}}{\bar{x}} \tag{9}$$

where n_p is the number of samples (processors), and

$$\bar{x} = \frac{\sum_s^{n_s} x_s}{n_s} \tag{10}$$

is the mean value. CTSP supports generating this data for any surface property by simply prepending `cv_` to the variable name in the output command variable list. Figures 6(c) and 6(d) plot this coefficient for the results shown above. We can observe that while in both cases, surface deposition variation is within 0.01 σ near the vent, the deposition in the region by the thrusters remains noisy, with variation approaching 1 σ in many instances. Our confidence in results from this region is greatly increased by moving to the higher number of processors. In order to maintain consistency in the timing study, the macroparticle weight was adjusted as $n_p w_{mp,1}$. This scaling maintained the total number of simulation particles across the entire MPI universe constant. As can be seen in Figure 5, MPI processing results in a superior parallel efficiency compared to multithreading. Interprocess communication is limited to sending global statistics such as the number of particles to the root rank. In addition, communication is necessary to generate intermediary result plots for animation, but this output was not activated for this timing study. Therefore, each process runs effectively independently, and as such, we would expect uniform and near perfect efficiency. This is clearly not the case, as the efficiency drops to 66% with 6 cores. The reason is two fold. First, the simulation shares the operating system with many other processes, and with all cores being utilized, some processes will need to be temporarily suspended to give other threads a chance to run. Secondly, the 6 cores share the same bus to access data in RAM. It is common to observe memory intensive simulations become *bandwidth limited*, with the CPU remaining idle while needed data is fetched from memory. High end CPUs, such as Intel Xeon, attempt to alleviate this limitation by including a larger cache.

3. PROGRAM FLOW

3.1 Input File

Various changes have also been made to the overall program flow. CTSP simulation follows a list of commands (or operations) specified in a text input file. These commands have the following structure:

```
operation_name{key1:value1, key2:value2, ...}
```

where each value can take a form of a single number or a string, or a list of values enclosed within square brackets. Furthermore, the list values can themselves be `value@time` tupples, with the code utilizing linear interpolation for intermediary data. For instance, the following command uses Regular Expression to assign a linearly decreasing temperature on all groups with names such as “radiator1”, “radiator2”, and so on.

```
surface_props{comp_name:/radiator\d+/, temperature:[250@0, 200@300]}
```

This functionality has existed in the code for some time. However, until recently, all operations had to be specified in a single file. This made the code inflexible for parametric studies or cases consisting of a sequence of simulations, as encountered during deployment or servicing activities. The latest code version supports splitting the input among multiple file by defining a new `include` command to insert contents of another file. Furthermore, the input file can contain variables which are either substituted directly or evaluated as mathematical expressions. This makes it possible to define a “primary” file outlining the simulation steps. This file is shared among simulations. Each simulation is started with a unique “driver” file that mainly sets simulation specific variables. For instance, we can have

```
set_variable{name:vent_flux, value:1e-12}
include{ctsp-main.in}
```

with the main file consisting of

```
surface_load_unv{...}
molecular_mat{name:hc1, ...}
source_cosine{comps:small_vent, mats:hc1, flux:$flux$}   #direct substitution
source_cosine{comps:large_vent, mats:hc1, flux:%4*flux%} #expression evaluation
run_sim{...}
surface_save_vtk{...}
```

The actual flux values are substituted based on the definition in the driver file.

3.2 Time Integration

The simulation input file now also supports loops and time integration. We are often interested in simulating contaminant transport over a long time interval, such as an entire thermal-vacuum test campaign, deployment, or even the expected mission lifetime. Due to the typical complexity of a surface model and the need for a large number of particles to reduce noise, simulations typically run two or even three orders of magnitude slower than real time. Since typical mission life times span multiple years, attempting to directly simulate the entire mission is clearly not feasible. This is also true for ground based testing, which may take weeks.[6] Even at only a single order of magnitude ratio of real to simulated time, modeling the entire test campaign would take almost a year.

Therefore, historically we have used the code to compute steady-state surface flux Γ_0 and then used some analytical scaling model to integrate the value over the entire lifetime. For instance, using a popular exponential model with rate decay by a factor of $1/e$ every τ hours, we have

$$m_{surf} = A \int_{t_0}^{t_f} \Gamma_0 \exp\left(-t/\tau_{hrs}\right) dt \tag{11}$$

with t_f being the end of life time, measured in hours.

This historical approach is acceptable in situations in which the transmission coefficient from the source to the target surfaces remains constant. In other words, surface temperature needs to remain constant, since it is the factor driving molecular re-emission. For this reason, a new functionality has been implemented to allow performing long-term integration directly within the code. CTSP can now be run as a sequence of short-steady state simulations, with deposited mass scaled by some constant user specified value between each step. In future code iterations, mathematical models such as the one given in Equation 11 will be supported. An example input file configuration is listed below.

```
world{set_time:3d}
run_sim{dt:1e-4, num_ts:1000, ts_steady_state:500}
scale_outgassing{mats:hc1, factor: 12.34}
world{set_time:3d2h}
run_sim{dt:1e-4, num_ts:1000, ts_steady_state:500}
scale_outgassing{mats:hc1, factor: 10.22}
world{set_time:3d4h}
...
```

The scaling factors take into account the ratio of real-time duration between steps (2 hrs) to the actual steady-state simulated time ($500 \cdot 10^4 = 0.5$ s), as well as any outgassing decay exhibited during this time interval. In cases where the factor remains constant, the above algorithm can be simplified using newly implemented blocks,

```
world{set_time:3d}
block_begin{repeat:5, expression:1}
run_sim{dt:1e-4, num_ts:1000, ts_steady_state:500}
scale_outgassing{mats:hc1, factor: 12.34}
world{add_time:2h}
block_end{}
```

The block repeats the given number of times as long as the expression evaluates to a positive value. Support for indexing into arrays remains for future work.

3.3 Data Input and Output

Improvements have also been made to the input and output capability. Historically, CTSP was capable of loading surface geometries using the following mesh formats: Universal, Abaqus, INPF, OBJ, and STL. Data could be exported in Universal, STL, Tecplot, and VTK (Paraview) formats. Mainly in support of client activities, several additional loaders have been added. These include a loader for the Cart3D .tri format and VTK unstructured grid files.

In addition, support for loading `.tssgm` TSS geometry files has been greatly expanded. TSS (Thermal Synthesizer Software)[7] is a popular thermal analysis tool routinely used to model radiative and conductive heat transfer on space systems. Due to the legacy use of thermal black body "form factors" in performing molecular transport analyses, it is commonplace for contamination engineers to use the same surface models used by thermal designers. The thermal models also offer the benefit of capturing sufficient amount of detail without overwhelming the simulation with small features such as screw holes. Unlike standard FEM formats that store a surface mesh composed of nodes and cell connectivity, TSS uses analytical objects, such as cones and bricks, to define the surface geometry. Each individual shape can be translated or rotated about its primary axis, and shapes can also be grouped into a tree of assemblies, each possibly having its own transformation or mirroring. Furthermore, these transformations can be specified using mathematical expressions utilizing named constants. The listing below shows a typical TSS file structure

```
assembly  MGSE.1
  units = meters
  mirror = "NONE"
  rot1 = x,"0.00" rot2 = y,"0.00" rot3 = z,"-1*DEP" tx = "-0.44" ty = "-1.64" tz = "0.00"
  assembly  bracket.1
    units = meters
    mirror = "NONE"
    rot1 = x,"0.00"  rot2 = y,"0.00"  rot3 = z,"0.00"  tx = "0.00"  ty = "0.00" tz = "0.00"
    sphere n_sphere
      units = meters
      param = "0.01", "-0.01", "0.01", "0.00", "360.00"
      active = OUT sides = SINGLE submodel = MAIN include = RADK,CC
      initial_id = "45282"
      rot1 = x,"-0.00" rot2 = y,"0.00" rot3 = z,"0.00" tx = "-0.20" ty = "2.07" tz="1.2"
```

While CTSP contained rudimentary support for TSS for some time, it was only very recently that the loader was made sufficiently robust to support transformations, variable substitutions, and the majority of TSS shapes. Some features, such as boolean operations, are not yet implemented. Besides the need to develop a parser capable of processing the complex input file and evaluate the embedded mathematical expressions, we also had to write code to tessellate additional shapes. TSS analytical objects are converted into a collection of triangles and/or quadrangles. This causes some particular challenges when dealing with conical sections. We sometimes encounter situations where a disc, which would be treated in TSS as a completely smooth object, does not fully cover an opening due to the jagged edges effect arising from a finite number of θ sections. This can occasionally introduce non-physical sneak paths that would not be present in the native TSS formulation. The current workaround is

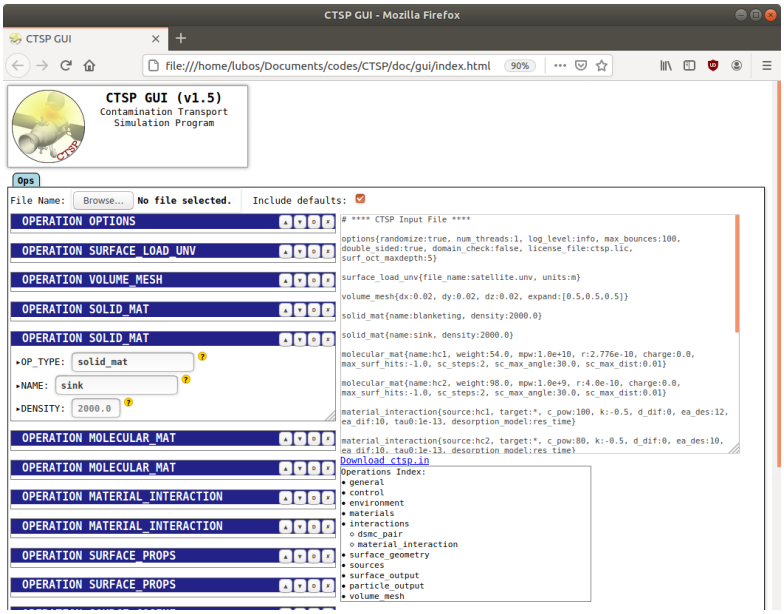


Figure 7. HTML-based GUI for generating and editing simulation input files

to utilize a corrections file to specify custom fine tessellation on specific elements. Related to this ability to use thermal surface models, CTSP can now also load surface element temperature time histories in SINDA format. These temperatures can be used along with, or in lieu of, the temperature profiles specified at the component group level. Grouping of data by TSS node id has also been added to the restart functions to allow continuing a solution over multiple TSS geometry files.

3.4 GUI

While a fully integrated GUI is still pending, CTSP now ships with an HTML-based environment for generating input files. This GUI can be run locally by opening the HTML file in a web-browser. The GUI uses a text-based database file to store all known operations and their parameters. The list of available commands is loaded as drop down options for a text field created by clicking an "Add Operation" button. The operation is then loaded with initial values.

We had some prior experience developing similar HTML-based solutions, and selected this route since web browsers are ubiquitous. Since the user interface is generated within the browser, there is no need to utilize 3rd party cross-platform user interface libraries (such as QT), nor the need to develop a custom solution for each target operating system. There are however some downsides to the current approach. Due to security concerns, HTML applications have only a limited access to the local file system. They also cannot launch local programs. As such, it is impossible to use a web-browser based GUI to start the simulation, at least not when running locally. Monitoring the run by periodically reading a log file and plotting time histories is also difficult unless the result file exists in a location that is deemed as safe to access by the web browser. This access path is not universal across browsers and operating systems.

4. MOLECULAR MODEL

4.1 Material-Material Interactions

CTSP nominally uses residence time

$$\tau_r = \tau_0 \exp\left(\frac{E_{a,des}}{RT}\right) \tag{12}$$

to determine whether a molecule sticks to a surface. Here $\tau_0 = 10^{-13}$ s is a scaling factor, $E_{a,des}$ is the activation energy of desorption, R is the gas constant, and T is the surface temperature. Earlier code versions allowed the

user to specify only a single value of activation energy for each molecular species. As such, it was not possible to distinguish between, let's say, a water molecule interacting with a bare metal substrate and other water molecules already adsorbed on the surface. The model has been updated to allow specifying two unique activation energies: energy of sublimation and energy of vaporization. The submlimation energy is used for molecules interacting with the substrate material. This energy can be specified uniquely for each flying – solid material pair. The vaporization energy is used for molecules interacting with the second or subsequent surface layers. This energy is assumed to be independent of the neighbor population. This approach thus implements some features of the BET model,[8] but without taking into account the localized accumulation sites. CTSP simply keeps track of the number of molecules of each flying material species that have adsorbed to the surface. The code assumes that these molecules instantaneously mix into a "soup" with no spatial concentration gradients.

When a molecular particle impacts a surface, the code first selects the active monolayer from area ratios. CTSP approximates molecules as spheres of some user-provided radius. The total area covered by the adsorbed molecules is

$$A_{sl} = \sum_s n_{p,s} \pi r_s^2 \tag{13}$$

The sum is over material species, and $n_{p,s}$ is the number adsorbed molecules of species s. r_s is the molecular radius. The number of monolayers is

$$N_{mL} = A_{sl}/A_{ele} \tag{14}$$

where A_{ele} is the surface element area. If $N_{ml} \geq 1$, or if $N_{ml} < 1$ and $\mathcal{R} < N_{ml}$, the impacting molecule is assumed to contact another molecule, and the vaporization energy $E_{a,vap}$ is used. Otherwise, the target material is randomly selected from the substrate, which can consist of a heterogeneous composition. The corresponding sublimation energy is used in this case. Similar approach is used when modeling surface desorption. For surface elements containing multiple monolayers, the evaporation is limited to a single layer.

4.2 Saturation Vapor Pressure Treatment

While the residence time model, Equation 12, is commonly used in contamination modeling,[9] it is not always apparent how to correlate it to experimental measurements. Furthermore, activation energies for commonly encountered materials are not easily available. On the other hand, saturation vapor pressure curves for many common chemicals can be found. These curves are often given in terms of coefficients of the Antoine Equation,

$$\log(p) = A - \frac{B}{C+T} \tag{15}$$

with the log typically being base 10. Various additional fits also exist. For instance, Murphy and Koop[10] provide a detailed review, along with their own fit, of water ice vapor pressure,

$$p_{ice} = \exp\left(9.550426 - \frac{5723.265}{T} + 3.53068\ln(T) - 0.00728332T\right) \tag{16}$$

This model has been implemented using Langmuir flux to estimate the number of molecules generated in a time step Δt on a surface element with area A as

$$N_{cap} = \frac{p}{\sqrt{2\pi k_B T m}} A\Delta t \tag{17}$$

Here m is the molecular mass, and k_B is the Boltzmann constant. This approach assumes expansion to vacuum. Equations 15 and 16 assume evaporation from a sufficiently large reservoir. It is possible for the surface layer to contain fewer molecules than predicted by N_{cap}. Therefore, the actual number of molecules desorbed from the surface element is

$$N_{gen} = \min(N_{cap}, N_{surf}) \tag{18}$$

The evaporation capacity is then updated to reflect the released matter,

$$N_{cap} - N_{gen} \rightarrow N_{gen} \tag{19}$$

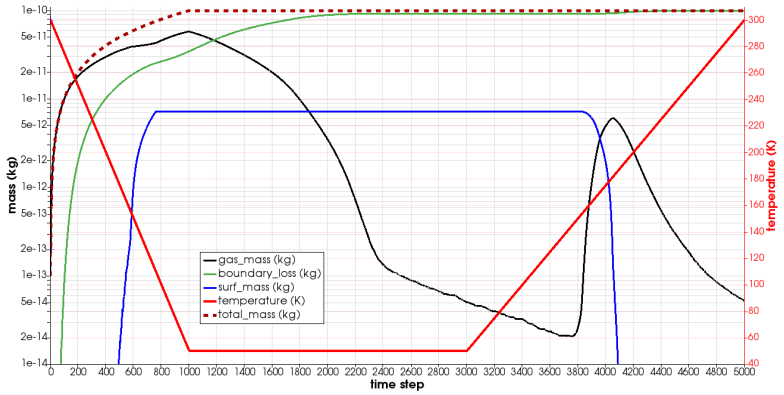


Figure 8. Log of temporal mass distribution for the water desorption example

For consistency, this same model is also used to characterize molecular sticking. On surface impact, we compute effective sticking coefficient from

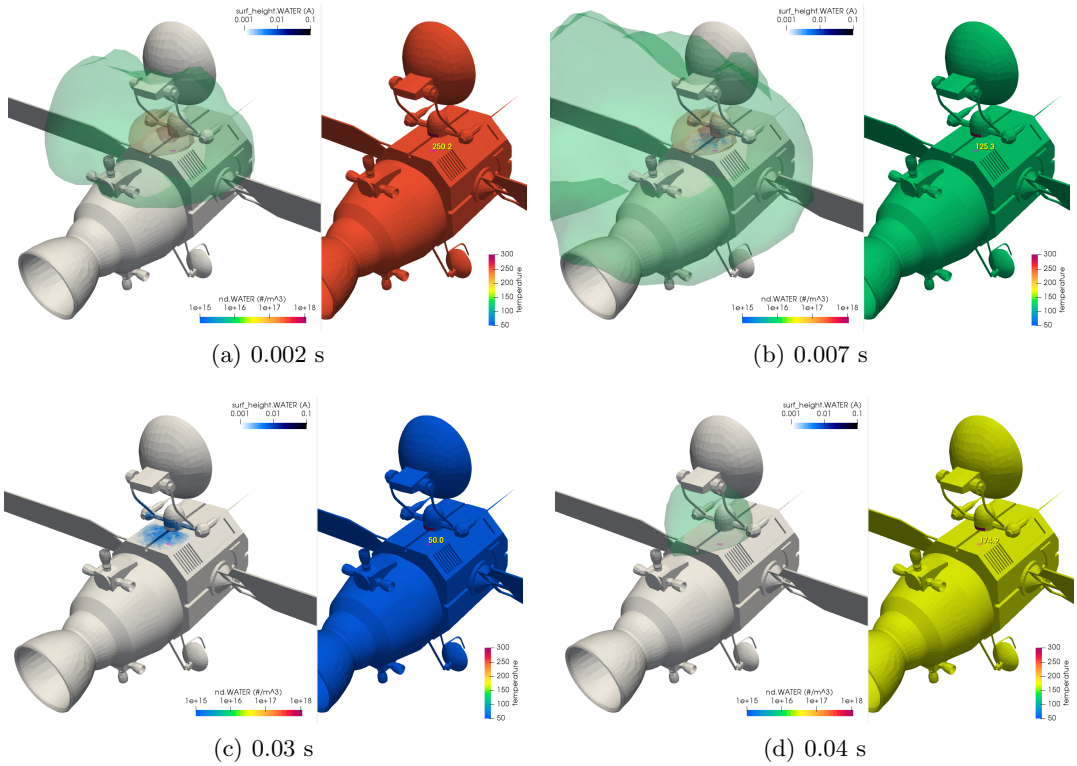
$$c_{stick} = 1 - \frac{N_{cap}}{w_{mp}} \qquad c_{stick} \in [0, 1] \tag{20}$$

4.3 Variable Weight

Actually implementing the above model in a particle code such as CTSP is complicated by the use of simulation "macroparticles", with each corresponding to some large number of real molecules. From the ideal gas law $p = nk_bT$, we can compute that there are over 10^{16} molecules per cubic meter at $T = 300$ K even at the limiting $p = 10^{-6}$ Torr pressure found in thermal vacuum facilities. Density near vents may exceed this pressure by several orders of magnitude. Therefore, we can quickly estimate that macroparticle weight will be of $O(12)$ for a typical simulation utilizing about million particles and a domain spanning few tens of m^3. Adsorption of this single simulation particle onto a surface increases that surface element's deposition thickness by $O(-3)$ Å, assuming surface element area of $(1\text{ cm})^2$ and contaminant density of 1 kg/m^3. This deposition thickness corresponds to the minimum resolution in the solution, given the above assumptions. It may not be acceptable to studies attempting to resolve very small contamination levels. Furthermore, in simulations capturing dynamic behavior such as temporal decay in vent rates, the macroparticle weight appropriate for the initial high flow rate case may not result in the vent producing many, or any, simulation particles once the flow rate decreases.

Therefore, the current implementation also utilizes a variable weight approach, in which a user specified number of simulation particles is generated on each surface element either from desorption or from other sources. At the same time, in order to prevent introduction of particulates with an insignificant weight, we also specify minimum w_{mp}. If the generated particles would have weight smaller than the prescribed value, we first attempt to create a smaller number of particles with $w_{mp,min}$. The remaining molecular count is collected into a storage container on each surface element, and at each time step, we attempt to sample a particle with probability $N_{collected}/w_{mp,min}$.

To demonstrate this and the partial pressure models, Figure 9 shows results from a simulation in which we let the vent on the example satellite generate water vapor at a constant mass flow rate for 1000 $\Delta t = 10^{-5}$ s steps. The source is subsequently turned off. At the same time, the surface temperature decays linearly from 300 K to 50 K. The surface remains at this temperature until time step 3000 ($t = 0.03$ s). It subsequently linearly warms up over the next 2000 time steps. Figure 8 plots the temporal evolution of surface mass, mass in the gaseous phase, and the cumulative mass that has left the domain through the boundaries. The surface temperature is also shown. Figure 9 plots the corresponding surface data. As expected, we initially observe the formation of a gaseous plume, but no accumulation on the surface, see Figure 9(a). Later, once the surface cools to approximately 130 K, Figure 9(b), water ice begins to accumulate. With the cryogenic surface and the source turned off at $t > 0.01$ s, the water ice remains trapped on the surface. This can be seen from the horizontal profile of the blue surface mass line in Figure 8. Once the surface warms up again to around 130 K, we start observing a dip in this curve and a corresponding increase in the gaseous population, the black line. This is



(a) 0.002 s (b) 0.007 s

(c) 0.03 s (d) 0.04 s

Figure 9. Progression of water surface accumulation and gas water density with linearly varying surface temperature

demonstrated in the surface plot, Figure 9(d), by the appearance of a low density gas plume, and the removal of the adsorbed material.

4.4 DSMC

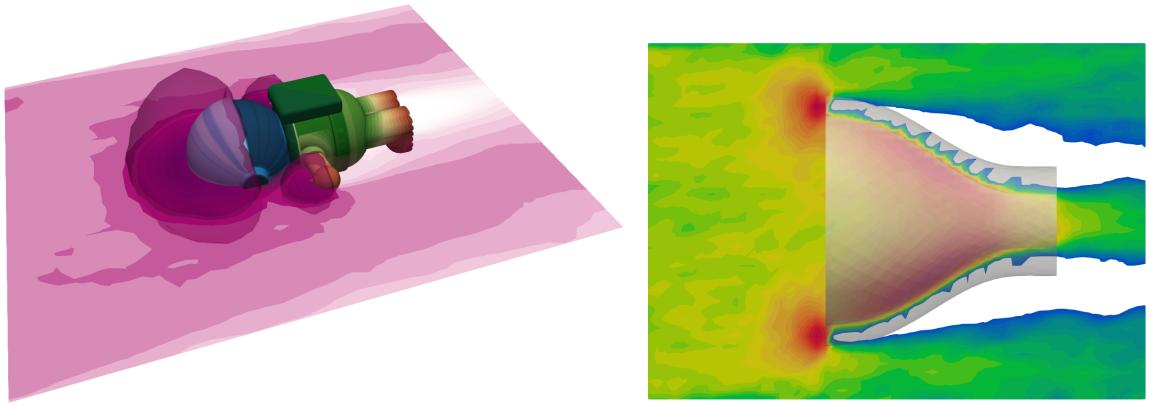
Although most applications of contamination transport involve low-density free molecular flows in which intermolecular interactions can be ignored, there are some instances in which this assumption is not valid. Some examples include vacuum chamber repressurization, and the initial venting of enclosures after exposure to vacuum. Rarefied gases, in which collisions are frequent enough not to be ignored, but not frequent enough to assume velocity distribution function thermalization, are routinely simulated with a technique called Direct Simulation Monte Carlo.[11] This scheme, based on the No Time Counter (NTC) and Variable Hard Sphere (VHS) molecular model, has been implemented in CTSP. DSMC simulations are the only instances in which the volume grid is used to advance particles. DSMC computes collisions in volume cells, with all particles located within the same cell being able to participate in the collision. Figure 10 shows two examples from a DSMC simulation. The first one modifies the typical "flow past a sphere" test case to a more complex geometrical shape. The astronaut models was obtained from GrabCad.com[†]. The second figure shows gas density for a uniform flow encountering a converging nozzle.

5. PARTICULATES

5.1 Subcycling

Various improvements have also been made to the particulate model. In the absence of electrostatics, forces acting on molecules are small enough to be safely ignored. This is not the case with the much larger particulates, and incorporating aerodynamic drag and gravity is critical to simulations of clean room environments and spacecraft fairings pre-launch environments. Since acceleration $d\vec{v}/dt$ scales with forces, which in turn vary with particle position, it is critical to utilize small-enough time steps to correctly integrate the trajectory. Given the large

[†]`https://grabcad.com/library/spaceman-astronaut-1`



(a) Astronaut (b) Converging Nozzle

Figure 10. Example DSMC simulations: a) flow past a complex geometricy, b) gas ingestion into a converging nozzle

variation in velocities and characteristic spatial dimensions in typical contamination simulations, picking an appropriate Δt is not trivial. The time step is also very much particle type and location specific. Position of a particle following a spiraling streamline needs to be integrated with a much smaller time steps than a large particulate falling out in a predominantly still air. Therefore, particle push subcycling has been implemented. This feature is turned on automatically for particulates and can be optionally activated for molecules. With subcycling, we integrate particle velocity and position using two steps. First, we advance both $\vec{v}$ and $\vec{x}$ using the standard Leapfrog scheme, Equation 1. We next perform this same integration from the starting point using two half Δt steps,

$$\vec{v}^{k} = \vec{v}^{k-0.5} + 0.5\Delta t(1/m)\vec{F}^{k} \tag{21}$$

$$\vec{x}^{k+0.5} = \vec{x}^{k} + 0.5\Delta t\vec{v}^{k} \tag{22}$$

and

$$\vec{v}^{*} = \vec{v}^{k} + 0.5\Delta t(1/m)\vec{F}^{k+0.5} \tag{23}$$

$$\vec{x}^{*} = \vec{x}^{k+0.5} + 0.5\Delta t\vec{v}^{k+0.5} \tag{24}$$

Note that this integration scheme diverts from Leapfrog due to the forcing term no longer being evaluated at the integration mid-point. We next compare the resulting $\vec{x}^{*}$ and $\vec{v}^{*}$ to the $\vec{x}^{k+1}$ and $\vec{v}^{k+0.5}$ computed with the single Δt push. If the two positions and velocity vector directions are within tolerance, $\left|\vec{x}^{k+1} - \vec{x}^{*}\right| < \epsilon_x$ and $\left|\hat{v}^{k+0.5} \cdot \hat{v}^{*}\right| < \epsilon_v$, we accept the new position and move on to the next particle. Otherwise, we repeat this process with $\Delta t/2$ and if acceptable, move the particle using two steps. If not, we try again with four $\Delta t/4$ steps and so on, until a user-specified maximum number of of refinements is reached. Figure 11 shows this routine in practice. Here we consider a vacuum domain containing a hypothetical air pocket, denoted by the dots. In this region, there exists uniform velocity and density flow in the $+x$ direction. Without subcycling and using large time steps, it is possible for the particulate dropped from the source plate to completely miss the air pocket and impact the floor directly below the insertion area. Alternatively, there may be only a single step inside the air pocket, leading to some, but insufficient, axial displacement. Using smaller and smaller time steps, we eventually converge on the trajectory given by the black curve. Using subcycling, we recover this same trajectory, shown by the colored dashed line, even with a large time step specified in the input file.

5.2 Particulate Adhesion

The particulate surface interaction code in CTSP is quite limited. The post-impact velocity is computed from

$$v_2 = \alpha_{COR} v_1 \tag{25}$$

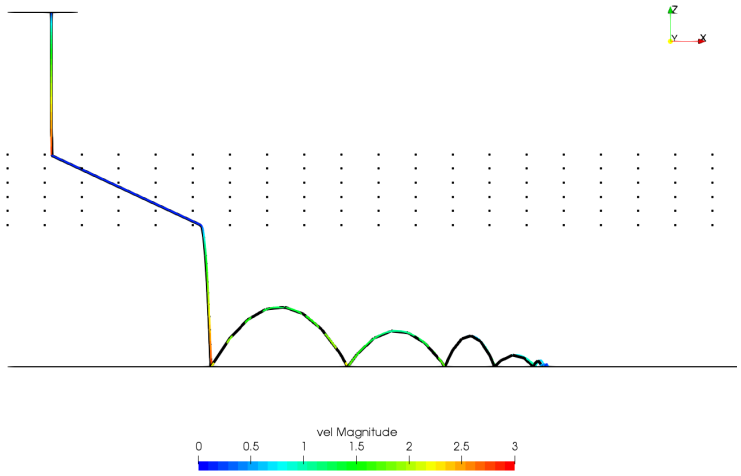


Figure 11. Comparison of particle traces with $\Delta t = 0.05$ s and 5×10^{-5} s.

where α_{COR} is a user supplied coefficient of restitution. The new direction is sampled by combining specular and diffuse reflection with the help of a coefficient of specularity, α_{spec}. Particles are loaded into the simulation by first computing the number of particles in several discrete bins, and then using the model of Klavins and Lee[12] to compute the fraction that actually leaves the surface.

This release model scales with a value of a characteristic acceleration magnitude. It is not clear how to properly account for external environmental effects, such as vibration and aerodynamic drag encountered during launch or on-pad environmental control system operations. This method also does not take into account the effect of surface charging, which may be more prevalent in particulates exposed to sunlight. We have therefore started working on implementing a new model based on ideas from the Discrete Element Method (DEM)[13] as well as a prior work of Crook.[14] With this scheme, we assume that there exists a spring-like attractive force F_{surf} that keeps particulates attached to the surface,

$$\vec{F}_{surf} = k_{surf} \left(\vec{x_0} - \vec{x}_p \right) \tag{26}$$

where $\vec{x}_0$ is the attachment location corresponding to either the initial loading location or the point of surface impact. We also let the external environment introduce vibrational (modeled as a force acting parallel to the surface), aerodynamic (computed from the drag equation), and electrostatic (utilizing the electric field arising between a charged plane and a charged sphere) forces. The total is then used to update the particulate relative velocity, and the displacement from the surface. Particles that travel distance exceeding some threshold are separated from the surface. This model introduces additional physical model, also requires setting coefficients, such as the effective spring constant k_{surf}. This value could be conceptually obtained by attempting to correlate results to the Klavins and Lee model, or from laboratory observations of particulate detachment.

6. EXPERIMENTAL VALIDATION

We have also initiated effort to perform experimental validation. Unfortunately, duo to the on-going COVID-19 pandemic, this effort has not yet been completed and remains as future work. In this section we describe the experiment plan. Our goal is to validate both the molecular and the particulate models.

6.1 Molecular Transport

Molecular analyses, in general, assume that molecules move in straight lines, and upon surface impact, reflect in a random direction following the cosine law angular distribution about the normal direction. However, we are not aware of much recent work to actually experimentally verify this hypothesis in context of contamination transport, outside of the legacy MSX flight experiment.[15] Therefore, in partnership with Prof. Joseph Wang at

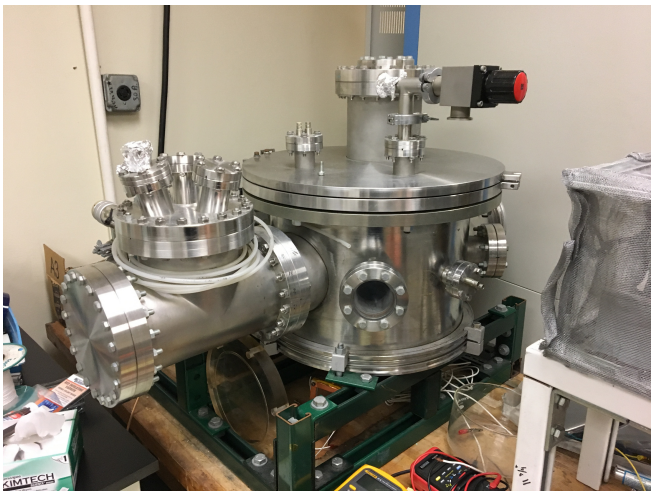


Figure 12. Vacuum chamber to be used for a future experimental validation

USC, we started initial effort to setup a molecular transport experiment utilizing a small vacuum chamber, Figure 12, and a Faraday TQCM made available under a loan from NASA Goddard Space Flight Center. The test plan calls for the TQCM to be placed at varying viewfactors from an outgassing test article, including a direct line of sight, behind a baffle, and outside a bakeout box. Given the simplicity of the setup, each configuration can be simulated numerically with a high degree of confidence in the modeled geometry description. Any discrepancies can then be attributed to deviation in molecular transport from the assumed model. Unfortunately, the campus shut down just as we were getting ready to start the first set of runs. We anticipate being able to resume the testing in the Fall 2020.

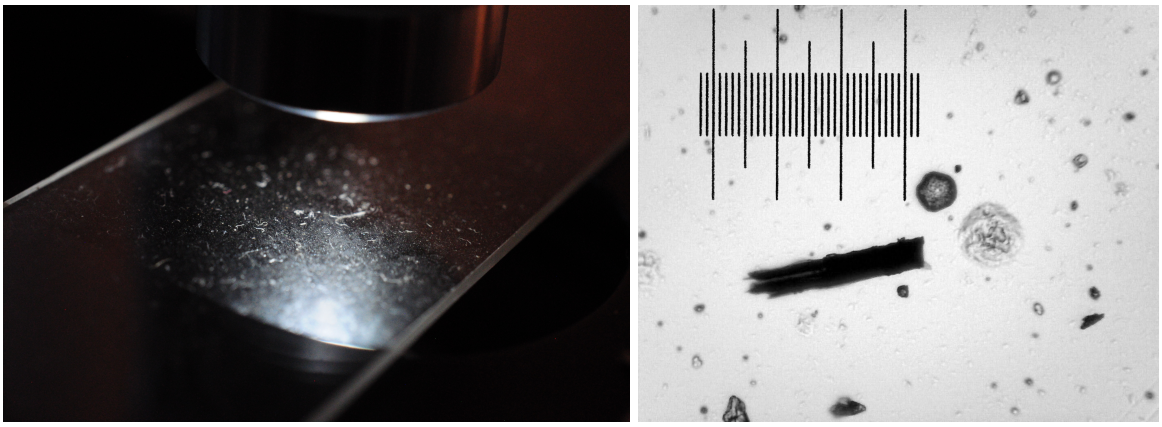
6.2 Particulate Characterization

We also began an effort to characterize particulate fallout using optical microscopes. Two instruments are being used: an $40-2500\times$ compound scope from AmScope equipped with a 1.3MP USB camera[‡], and a $220\times$ handheld 5MP USB digital microscope from AdaFruit[§]. Figure 13(a) shows a close up photograph of a microscope slide left exposed in an office environment for about 2 weeks. Particulate contamination is clearly visible. The image in Figure 13(b) characterizes these particulates when imaged through the compound microscope. Each ruler division corresponds to 10 μm. We can see a large number of sub-10 μm particles, followed by several particulates in the $25 \rightarrow 100$ μm range, along with one larger ≈ 250 μm particle.

Our plan is to use these instruments to characterize particulate fallout and transport under varying ambient conditions. For instance, we can image the final resting position of small grains dropped from a set height to approximate the coefficient of restitution. We can also utilize servo motors to apply rotational and translational motion to a sample. As a proof of concept, we attempted to characterize the effect of ambient air flow on particulate fallout, but the results are inconclusive. Figure 14(a) shows the experiment setup. Here a clean microscope slide was placed in front of a 5V computer fan. Another slide was placed on top of the fan, outside the produced air flow. The initial cleanliness of both slides was verified by taking a microscope baseline image, see Figure 14(b). The slides were left in an office environment for 12 hours, with the fan continuously operating. Figure 15 shows the resulting accumulation, with the top two figures corresponding to two spots on the "top" slide (the slide not exposed to the air flow). The two bottom images were sampled on the "bottom" slide exposed to the air flow. From a visual observation, it appeared that the bottom slide received a slightly greater accumulation of particulate contamination, contrary to our initial assumption. The likely explanation is that the ambient air, instead of preventing collection of particulates falling out from the space above the fan, actually increased the local concentration by moving a lot of dirty office air across the top surface of the slide. However, the fairly small difference in deposition indicates that a much longer test interval is needed.

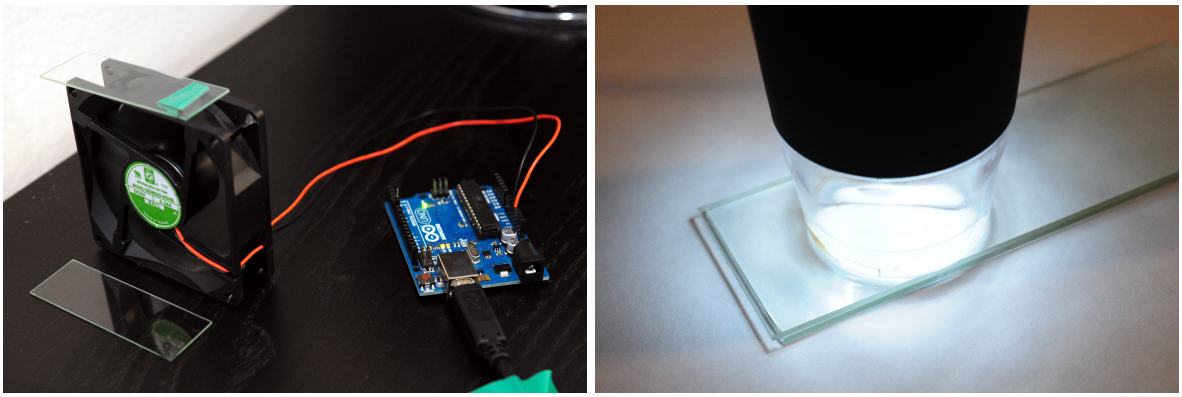
[‡] https://www.amscope.com/40x-2500x-led-digital-binocular-compound-microscope-w-3d-stage-1-3mp-usb-camera.html

[§] https://www.adafruit.com/product/636



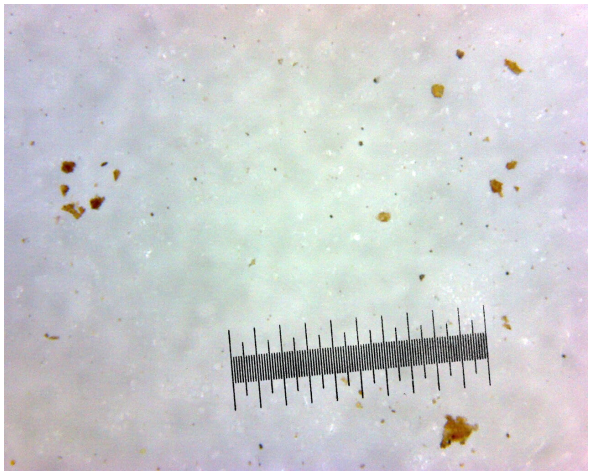
(a) Collected sample (b) Microscope Image

Figure 13. Photo of particulate matter collected on an exposed microscope slide (a) and observation through the optical microscope (b). Each ruler mark corresponds to 10μm.

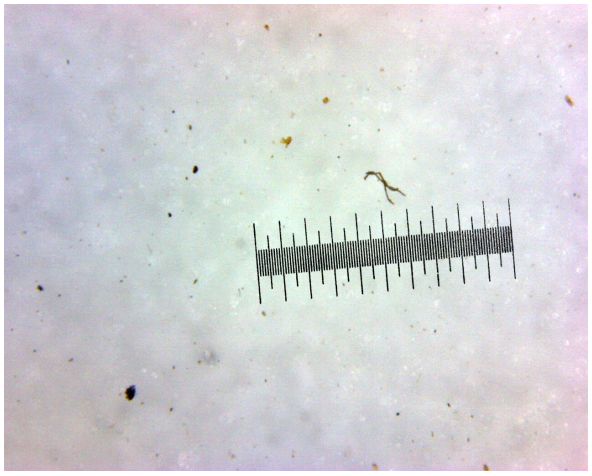


(a) Airflow experiment setup (b) Slide imaging

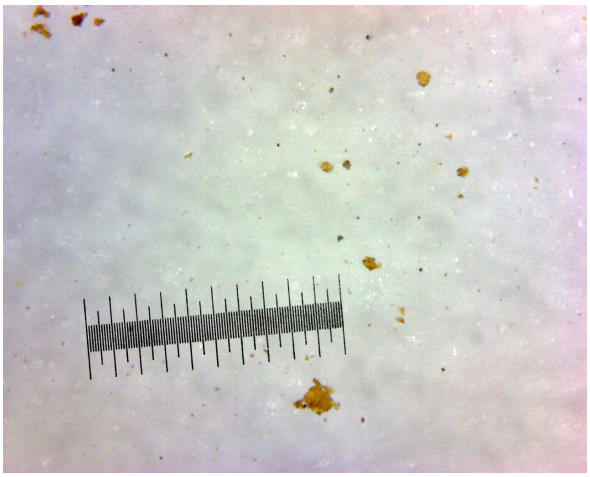
Figure 14. Experimental setup to characterize the effect of air flow on particulate collection.



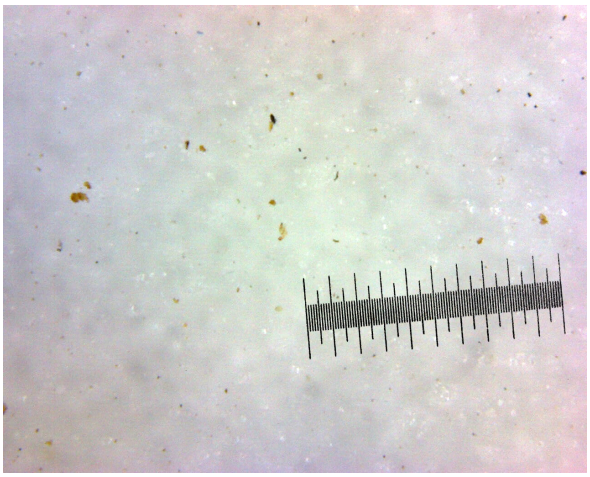
(a) bottom 1



(b) bottom 2



(c) top 1



(d) top 2

Figure 15. Images of dust particulates from the airflow experiment taken by the digital microscope

7. CONCLUSION

This paper summarized modifications made to the Contamination Transport Simulation Program (CTSP) since the most recent publication. We started by covering an optimization effort, which involved rewriting the octree used for the mesh-free particle tracing. We also discussed improvements to the input file structure, and introduced an HTML-based GUI for generating input files. We then discussed modifications to the molecular and particulate surface physics models, including the implementation of a desorption model based on partial pressures, and subcycling of particulate transport. The paper was concluded by an overview of an ongoing experimental campaign.

Acknowledgement

The authors would like to acknowledge Prof. Joseph Wang at USC for helping to facilitate access to a vacuum chamber as well as Eve Woolridge and George Meadows for providing a QCM collection system. We would also like to thank John Canham at Northrop Grumman and Michael Woronowicz at NASA Goddard / SGT Inc., for their insight and suggestions for improving the molecular adhesion model. John Chawner at Pointwise provided the surface mesh of the satellite model.

REFERENCES

[1] Brieda, L., "Numerical model for molecular and particulate contamination transport," *Journal of Spacecraft and Rockets* **56**(2), 485–497 (2019).

[2] Brieda, L., *Development of the DRACO ES-PIC code and fully-kinetic simulation of ion beam neutralization*, Master's thesis, Virginia Tech (2005).

[3] Jambunathan, R. and Levin, D. A., "CHAOS: An octree-based PIC-DSMC code for modeling of electron kinetic properties in a plasma plume using MPI-CUDA parallelization," *Journal of Computational Physics* **373**, 571–604 (2018).

[4] Wikipedia, "Z-order curve." `https://en.wikipedia.org/wiki/Z-order_curve`.

[5] Gropp, W., Thakur, R., and Lusk, E., [*Using MPI-2: advanced features of the message passing interface*], MIT press (1999).

[6] Rosecrans, G., Brieda, L., and Errigo, T., "MMS observatory TV results: Contamination summary," in [*28th Space Simulation Conference*], (2014).

[7] Panczak, T., Rickman, S., Fried, L., and Welch, M., "Thermal synthesizer system: an integrated approach to spacecraft thermal analysis," *SAE transactions* , 1851–1867 (1991).

[8] Brunauer, S., Emmett, P. H., and Teller, E., "Adsorption of gases in multimolecular layers," *Journal of the American chemical society* **60**(2), 309–319 (1938).

[9] Tribble, A. C., [*Fundamentals of contamination control*], vol. 44, SPIE Press (2000).

[10] Murphy, D. M. and Koop, T., "Review of the vapour pressures of ice and supercooled water for atmospheric applications," *Quarterly Journal of the Royal Meteorological Society: A journal of the atmospheric sciences, applied meteorology and physical oceanography* **131**(608), 1539–1565 (2005).

[11] Bird, G. A. and Brady, J., [*Molecular gas dynamics and the direct simulation of gas flows*], vol. 5, Clarendon press Oxford (1994).

[12] Klavins, A. and Lee, A., "Spacecraft particulate contaminant redistribution," 236–244 (1987).

[13] Cundall, P. A. and Strack, O. D., "A discrete numerical model for granular assemblies," *geotechnique* **29**(1), 47–65 (1979).

[14] Crook, R. A., "Estimating particle redistribution from launch loads," *Optical Engineering* **45**(5), 059001 (2006).

[15] Wood, B. E., Hall, D. F., Lesho, J. C., Dyer, J. S., Uy, O. M., and Bertrand, W. T., "Quartz crystal microbalance (qcm) flight measurements of contamination on the msx satellite," in [*Optical System Contamination V, and Stray Light and System Optimization*], **2864**, 187–194, International Society for Optics and Photonics (1996).