Quiet Direct Simulation (QDS):

Fast, Low-Noise Fluid Simulations

Scott Sikorski

Under Adam B. Sefkow with help from Mike Lavell

Department of Mechanical Engineering and Computer Science
University of Rochester with the UR Laser Lab for Energetics

Outline

- Our previous DSMC particle in cell (PIC) method and its problems
- The 1-D QDS algorithm base (in Python)
- The SOD Shocktube Problem solved using QDS
- Moving to multi-dimensional
- Enabling multi-threading and GPU computation with CUDA and NVIDIA GPU
- Where to go next
- Conclusion

DSMC Algorithm

At t = 0, we initialize our density of a cell giving each particle a mass of $f_o(x) / ppc$ on the mesh

We advance each particle using:

$x(t+\Delta t) = x(t) + N(0, 1) * \sqrt{(2 D \Delta t)}$

D:= Diffusion coefficient

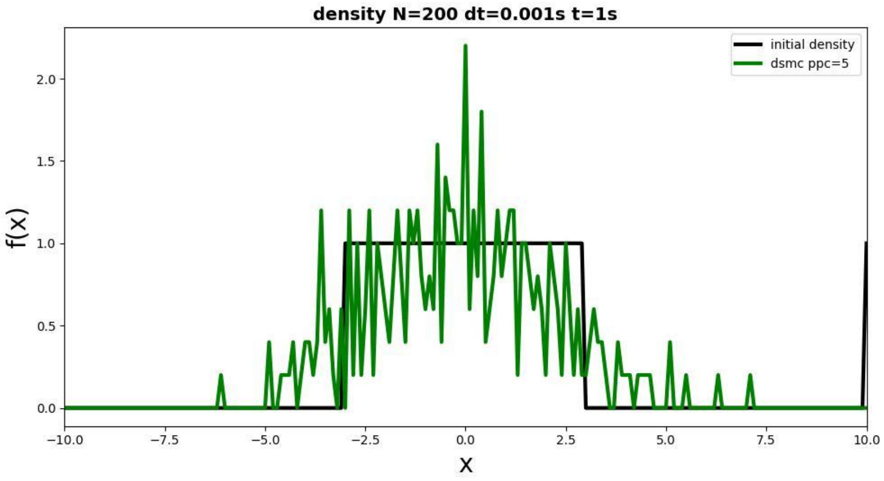
$N(0, 1)$:= Random normally distributed value with zero mean and unit variance

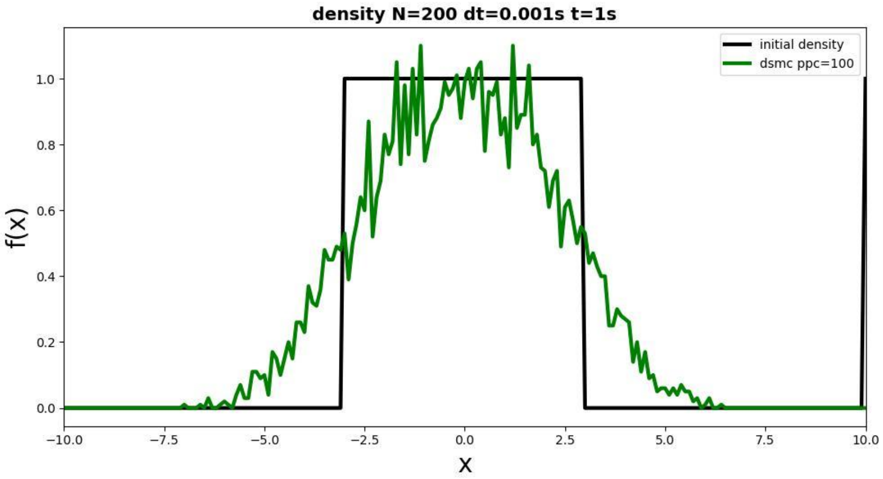
We finish a time step by reweighing our particle masses on the mesh to a cell, then repeat

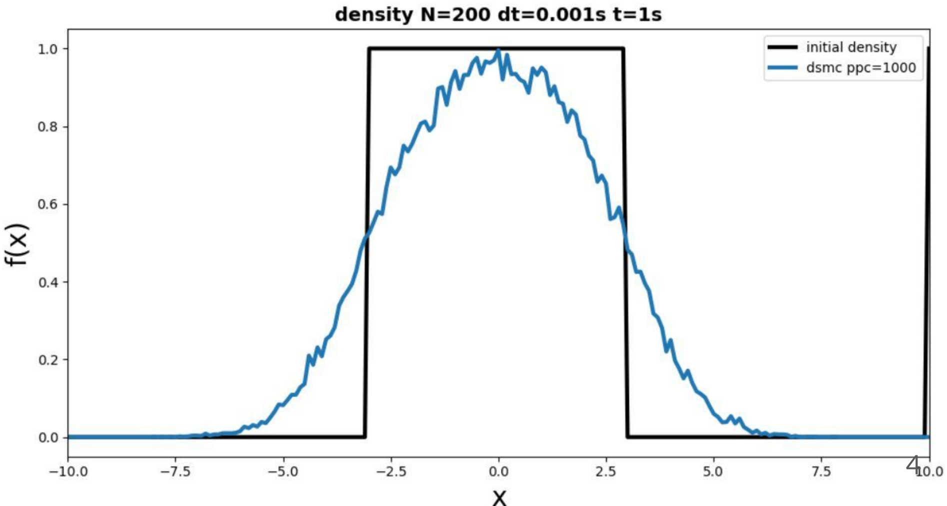
```
N:= Number of cells
mass:= Mass of a particle

for step in range(nsteps):
    particleCounter= np.zeros(N+1).astype(int)
    ran = np.random.normal(mean, variance, nParticles)
    for ip in range(nParticles):
       posn[ip] += ran[ip]*const
       cellIndex=mapPosnToCellIndex(posn[ip],dx,N,adjustOrigin)
       particleCounter[cellIndex] += 1

   dens= particleCounter*mass/dx
```

DSMC

Diffusion of a Slab







Some Limitations of DSMC

- Stochastic process leading to high levels of noise
  - Error with random sampling decreasing at 1/√(ppc)
  - Limited dynamic range for simulations
- Computationally expensive
  - Linear to number of total particles in simulation
- High memory usage
  - `N * ppc * sizeof(float) bytes for a grid`
  - Low cache hit rate

The QDS Algorithm (Without Euler Equations)

1. “Quietly” create the particles on the grid using the initial or previous time step grid moments

$$m_{ij} = \rho_i V w_j / (\sum_j^{ppc} w_j)$$

$$x(t+\Delta t) = x(t) + a_i \sqrt{(2 D \Delta t)}$$

2. Advance the particles and linearly distribute the mass values along the grid

$$m_{ij} = m_{ij} * W_{ij}$$

$$W_{ij} = \mathit{if}\ x_{i-1} < x_{new} \le x_i := (x_{new} - x_{i-1}) / (x_i - x_{i-1})$$

$$\mathit{if}\ x_i < x_{new} \le x_{i+1} := (x_{i+1} - x_{new}) / (x_{i+1} - x_i)$$

$$\mathit{else} := 0$$

3. Grid moments are calculated from advanced particles, put back on the grid, and then the particles are destroyed

$$\rho_i = m_i / V$$

w_j := Weight *j-th* value

a_j := Abscissa *j-th* value

$i \in \{1 \dots N\}$; $j \in \{1 \dots ppc\}$

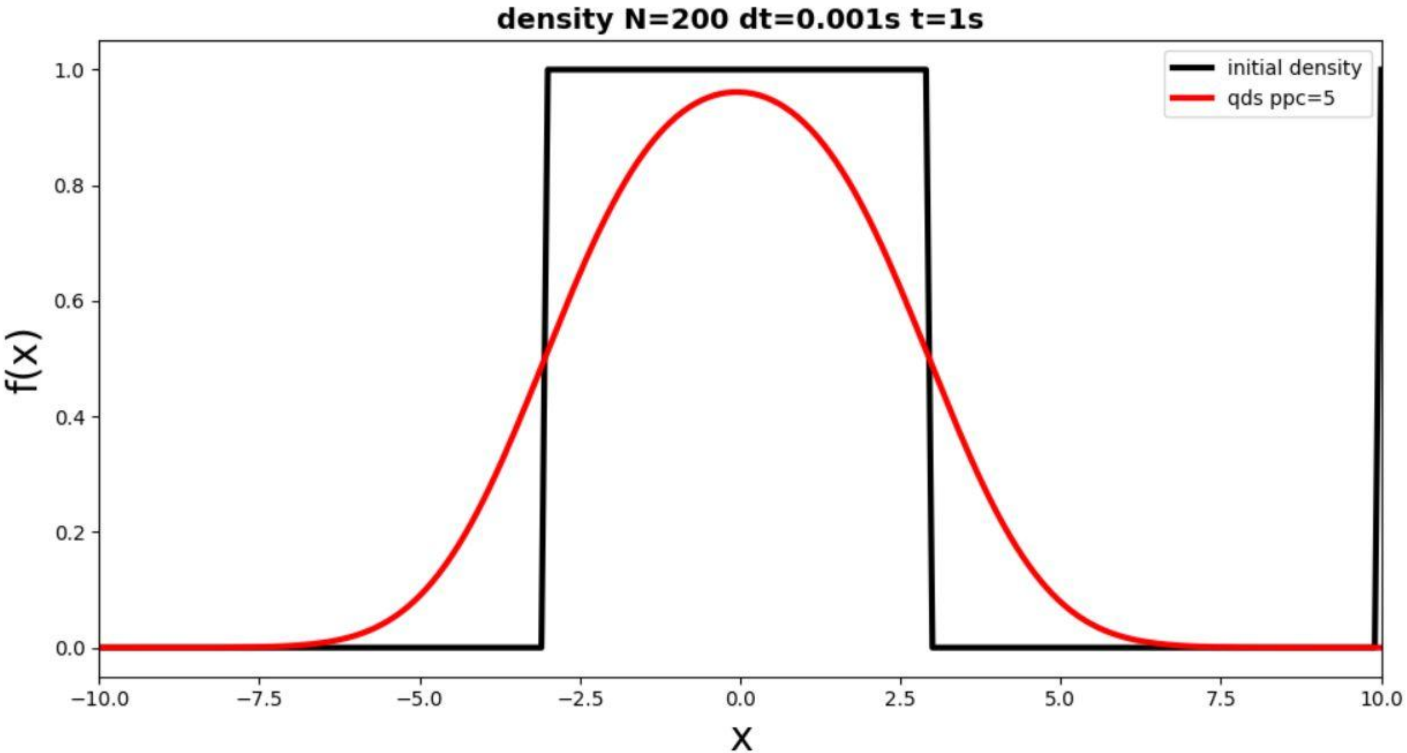
The QDS Algorithm (Without Euler Equations)

```
for step in range(nsteps):
    gridMass= np.zeros(N+1) # reset to zero
    # update particle position
    for ig in range(N):
        if dens_qds[ig] != 0.0:
            mass_qds= dens_qds[ig]*dx*weight/sum(weight)
            for ip in range(ppc_qds):
                xNew= xx[ig] + abscissa[ip]*const
                mycell= mapPosnToCellIndex(xNew,dx,N,adjustOrigin)
                if mycell>0 and mycell<N-1:
                   gridMass[mycell-1] += mass_qds[ip]*
                      linearWeighting(xNew, xx[mycell-2], xx[mycell-1], xx[mycell  ])
                   gridMass[mycell  ] += mass_qds[ip]*
                      linearWeighting(xNew, xx[mycell-1], xx[mycell  ], xx[mycell+1])
                   gridMass[mycell+1] += mass_qds[ip]*
                      linearWeighting(xNew, xx[mycell  ], xx[mycell+1], xx[mycell+2])
       dens_qds= gridMass / dx
```

Weight:= Hermite-Gaussian weight values for ppc

Abscissa:= Hermite-Gaussian abscissa values for ppc

The New and Improved Diffusion of a Slab



Why QDS?

- Computationally less expensive
- Precise values for mass, velocity, temperature, pressure, and more
    - And a low particle count (~4 particles per cell) is allowed due to elimination of Stochastic processes
- Large dynamic and flexible range for simulations
    - Kinetic, fluid, and plasma models can all be simulated
- Linear memory usage to the number of cells
- Ability to conduct reliable multi-dimensional simulations
- Ability to fully parallelize the algorithm

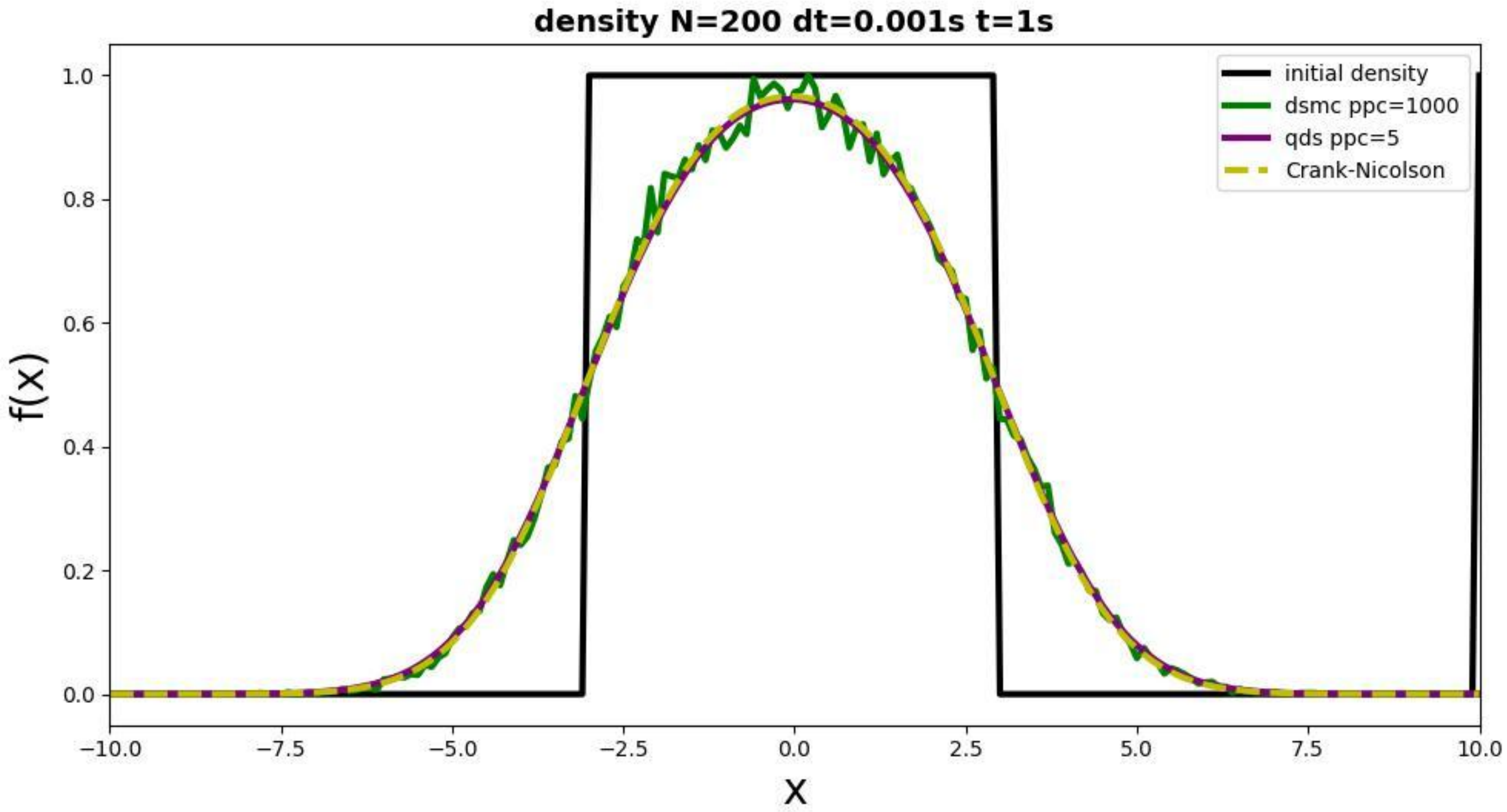
Runtime Performance

	Time for DSMC (s)	Time for QDS (s)	Factor Saved
Run 1	114.74	6.32	18.16
Run 2	114.37	6.29	18.17
Run 3	115.18	6.22	18.52
Run 4	125.06	6.01	20.80
Run 5	118.57	6.30	18.82
Run 6	120.34	6.55	18.37
Run 7	120.90	6.28	19.24
Run 8	121.34	6.12	19.84
Run 9	116.04	6.38	18.76
Run 10	122.76	7.27	16.89

- ppc (DSMC) := 1000
- ppc (QDS) := 5
- Diffusion coefficient:= 1.0
- N = 200
- t = 1s, Δt = 0.001s
- nsteps = 1000

Average time spent on DSMC
118.931s

Average time spent on QDS
6.3737s


density N=200 dt=0.001s t=1s
initial density
dsmc ppc=1000
qds ppc=5
Crank-Nicolson
f(x)
x
1.0
0.8
0.6
0.4
0.2
0.0
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

QDS Algorithm With Euler Equations

1. "Quietly" create the particles on the grid using the initial or previous time step grid moments

At position x_i on the mesh, each particle has an internal specific energy, (ε_i), which correlates to the degrees of freedom (γ value). Along with each particle's mass and velocity values.

$$\varepsilon_i = (d-1)\,\sigma_{vi}^{2} / 2$$

σ_v := Velocity variance

d:= Degrees of freedom = 3 for γ = 5/3

$$m_{ij} = \Delta x\, \varrho_i\, w_j / \Sigma\, w_j$$

$$v_{ij} = u_i + q_j \sqrt{}(2\sigma_{vi}^{2})$$

w_j, q_j := *j-th* weight and abscissa value for Hermite Gaussian, respectively

i ϵ {1 ... N}; j ϵ {1 ... ppc}

QDS Algorithm With Euler Equations

2. Advance the particles and linearly distribute the particle values onto the grid points

$$x_{ij}^{new} = x_i + v_{ij}\,\Delta t$$

$$\mathrm{m_i} = \Sigma_{\mathrm{j}}^{\mathrm{ppc}}\, \mathrm{m_j}\, W_{ij}$$

$$\mathrm{p_i} = \Sigma_{\mathrm{j}}^{\mathrm{ppc}}\, \mathrm{m_j}\, v_{\mathrm{j}}\, W_{ij}$$

$$\mathrm{E_i} = \Sigma_{\mathrm{j}}^{\mathrm{ppc}}\, \mathrm{m_j}\, (½\, v_{\mathrm{j}}^2 + \varepsilon_{\mathrm{j}})\, W_{ij}$$

$$W_{\mathrm{ij}} = \mathit{if}\ x_{i-1} < x_{new} \leq x_i := (x_{new} - x_{i-1}) / (x_i - x_{i-1})$$

$$\mathit{if}\ x_i < x_{new} \leq x_{i+1} := (x_{i+1} - x_{new}) / (x_{i+1} - x_i)$$

$$\mathit{else} := 0$$

QDS Algorithm With Euler Equations

```
for step in range(nsteps):
       massGrid = np.zeros(N+1)
       momGrid = np.zeros(N+1)
       enerGrid = np.zeros(N+1)
       for i in range(N+1):
           if (densGrid[i] != 0):
               for j in range(ppc_qds):
                   vPart = velGrid[i] + abscissa[j] * (2.0*varGrid[i])**0.5
                   mPart = densGrid[i] * dx * weight[j] / sum(weight)
                   ePart = (deg-1) * varGrid[i] / 2.0
                   xNew = xx[i] + vPart * dt
                   newCell = mapPosToCell(xNew,dx,N,xmin)

                   if (newCell > 0 and newCell < N-1):
                      weighting  = linearWeighting(xNew, xx[newCell-2], xx[newCell-1], xx[newCell  ])
                       weighting2 = linearWeighting(xNew, xx[newCell-1], xx[newCell  ], xx[newCell+1])
                       weighting3 = linearWeighting(xNew, xx[newCell  ], xx[newCell+1], xx[newCell+2])
                       distribute(newCell, mPart, vPart, ePart, weighting, weighting2, weighting3)
```

QDS Algorithm With Euler Equations

```
distribute(newCell, mPart, vPart, ePart, w1, w2, w3):
    massGrid[newCell-1] += mPart * w1
    momGrid[newCell-1] += mPart * vPart * w1
    enerGrid[newCell-1] += mPart * w1 * (0.5 * (vPart**2.0) + ePart)

    massGrid[newCell] += mPart * w2
    momGrid[newCell] += mPart * vPart * w2
    enerGrid[newCell] += mPart * w2 * (0.5 * (vPart**2.0) + ePart)

    massGrid[newCell+1] += mPart * w3
    momGrid[newCell+1] += mPart * vPart * w3
    enerGrid[newCell+1] += mPart * w3 * (0.5 * (vPart**2.0) + ePart)
```

QDS Algorithm With Euler Equations

3. Density, velocity, velocity variance, and other wanted moments are calculated from advanced particles, put on the grid, and then the particles are destroyed

$$\varrho_i = m_i / V$$

$$u_i = p_i / m_i$$

$$\sigma_{vi}^2 = (2E_i - u_i^2) / (m_i d)$$

$$T_i = ((E_i / m_i) - ½(p_i / m_i)^2) * ((\gamma-1)/R)$$

$$P_i = \sigma_{vi}^2 * \varrho_i$$

R:= Ideal gas constant (8.3145 [J/(mol K)])

QDS for Shocktube of large magnitude differences

<u>Initial conditions</u>

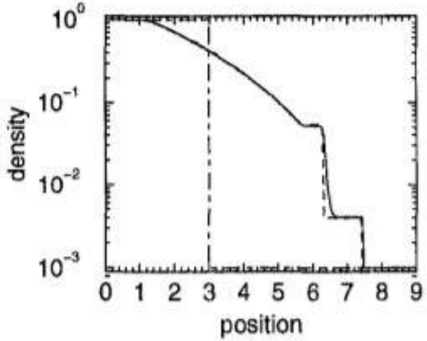
ϱ_L = 1.0 kg/m ϱ_R = 10^{-3} kg/m

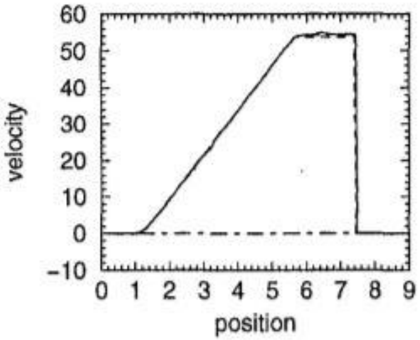
T_L = 10^3 K T_R = 10^{-3} K

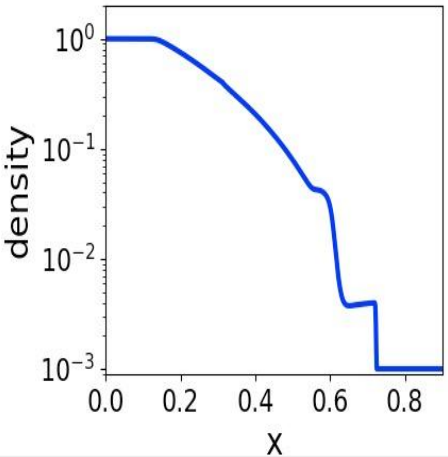
P_L = 10^3 Pa P_R = 10^{-6} Pa

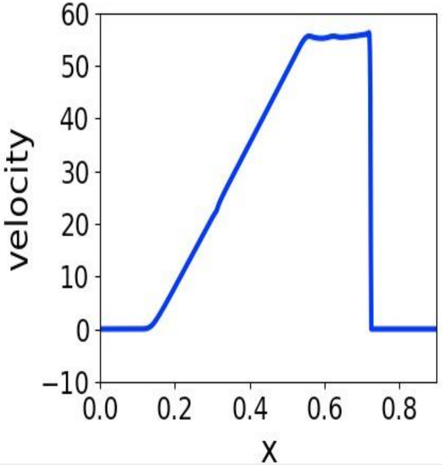
$v_{L,R}$ = 0 m/s

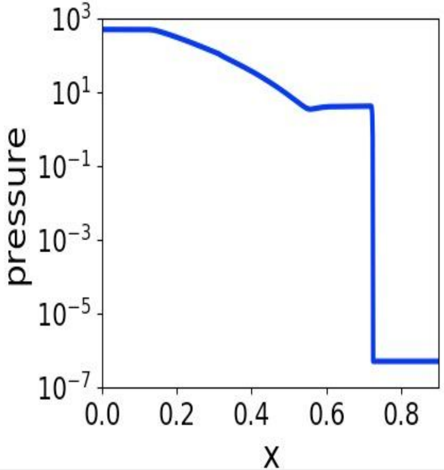
Left is split from right at x = 0.3m

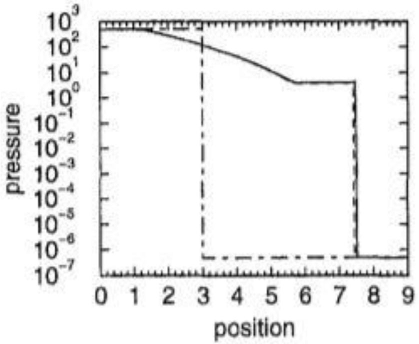












QDS for Shocktube of large magnitude differences

- Dynamic range can extensively be seen through a magnitude difference of 10^3, 10^6, and 10^9 for density, temperature, and pressure, respectively
    - The limits of this range have not been fully tested, yet continues to be accurate under these large magnitudes
- All these values are double-precision floating point
    - We can accurately replicate the simulation values within machine accuracy
- Simulation operates under unit-less magnitude
    - User is able to simulate macroscopic and microscopic processes

Some Limitations of QDS

- Limited Δt due to algorithm stability. Particles can't rapidly move across the grid so υ can't be too high
  - $\Delta t \leq \Delta x / \upsilon$
  - $\Delta t \leq \Delta x / \sqrt{(\sigma_\upsilon^2 + \upsilon^2)}$

- Can we find a way to have unconditional algorithm stability so that our higher dimensional simulations are limited only by physical accuracy?

- Currently limited to a first order numerical scheme
  - Truncation of the second-order and onward terms within the linear differentiation

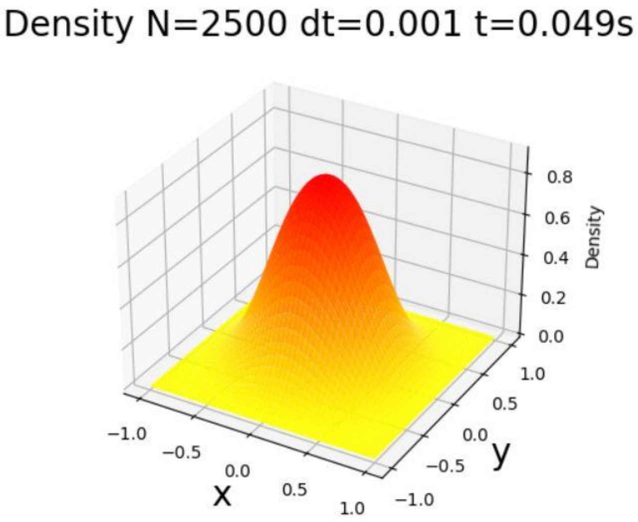
- Is expansion to second or third schemes computationally worth it?

Some Additional Features

- User specification of initial conditions
  - Density and temperature of the left and right side, split at the midpoint
  - Any simulation can take place within the Python script
- Simulations with normalized units for all values
- User molecule choice to run the simulation
  - Consists of the Z-value and the number of atoms that the user wants
  - Implemented using a stored hashmap that calculates molecule mass based on the number of protons, neutrons, and electrons of a given atom
  - Mass value is determined by this user specification

Moving to Multi-Dimensional QDS

- A new separate set of particles and a corresponding Hermite Gaussian quadrate is introduced for each dimension
  - J-D problems require ppc^J particles
  - This corresponds to an even greater computational save from DSMC
- We require 3^J operations to accurately linearly distribute our grid values
  - Comes from adjacent cell computations

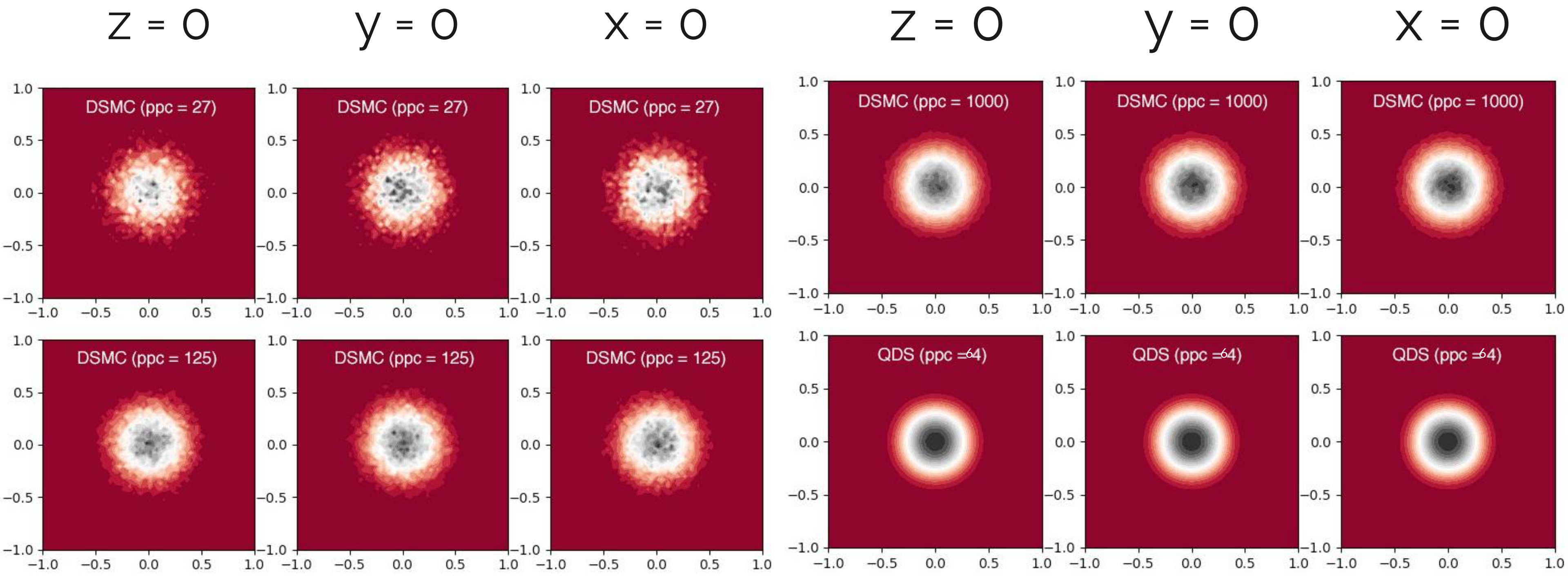


Segment of the 3D QDS implementation

```
if  (wz1!=0):
    if (wy1!=0):
        massGrid[xCell-1, yCell-1, zCell-1] += massPart * wx1 * wy1 * wz1
        massGrid[xCell  , yCell-1, zCell-1] += massPart * wx2 * wy1 * wz1
        massGrid[xCell+1, yCell-1, zCell-1] += massPart * wx3 * wy1 * wz1

    if (wy2!=0):
        massGrid[xCell-1, yCell  , zCell-1] += massPart * wx1 * wy2 * wz1
        massGrid[xCell  , yCell  , zCell-1] += massPart * wx2 * wy2 * wz1
        massGrid[xCell+1, yCell  , zCell-1] += massPart * wx3 * wy2 * wz1

    if (wy3!=0):
        massGrid[xCell-1, yCell+1, zCell-1] += massPart * wx1 * wy3 * wz1
        massGrid[xCell  , yCell+1, zCell-1] += massPart * wx2 * wy3 * wz1
        massGrid[xCell+1, yCell+1, zCell-1] += massPart * wx3 * wy3 * wz1
```

QDS vs DSMC - Diffusion of a 3-D sphere

Density slices



CUDA, Parallel Processing, and GPU Computation

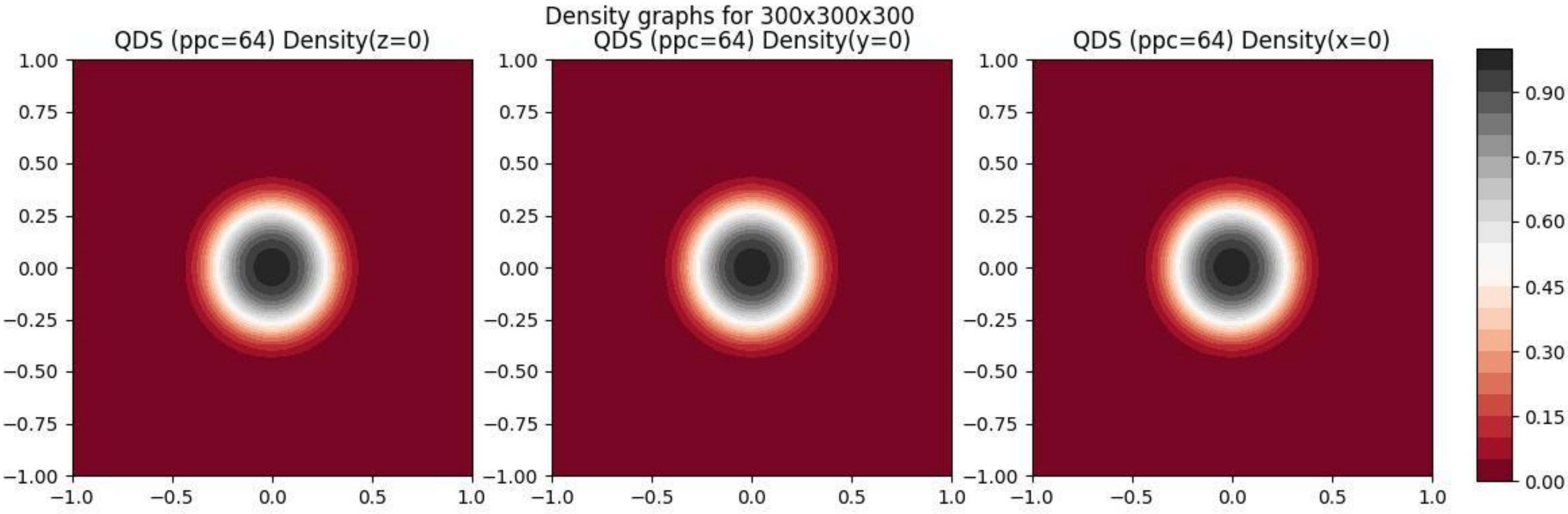
- With no change to the actual QDS algorithm and CUDA implementation of QDS, we are able to replicate our low-noise simulations
- Computing within Cuda kernels on NVIDIA GPUs has greatly decreased computational time
    - With no hand optimized code as of now, we are able to achieve our benchmark goal
        - We seek to be < 1 ns/particle/time step
        - We are achieving an average of ~1 ns/particle/time step for total computational time
- In general, multi-dimensional code is handled better
    - Each block represents a cell with threads being the particles

Main Code difference between Python/C++ and Cuda

```
if (wy1!=0 & wz1 != 0):
  massGrid[xCell-1, yCell-1, zCell-1]+=massPart*wx1*wy1*wz1
  massGrid[xCell,   yCell-1, zCell-1]+=massPart*wx2*wy1*wz1
  massGrid[xCell+1, yCell-1, zCell-1]+=massPart*wx3*wy1*wz1
```

```
atomicAdd(&massGrid[getOffset(xCell-1, yCell-1, zCell-1, xN, yN)],
    massPart*wx1*wy1*wz1);
atomicAdd(&massGrid[getOffset(xCell  , yCell-1, zCell-1, xN, yN)],
    massPart*wx2*wy1*wz1);
atomicAdd(&massGrid[getOffset(xCell+1, yCell-1, zCell-1, xN, yN)],
    massPart*wx3*wy1*wz1);
```

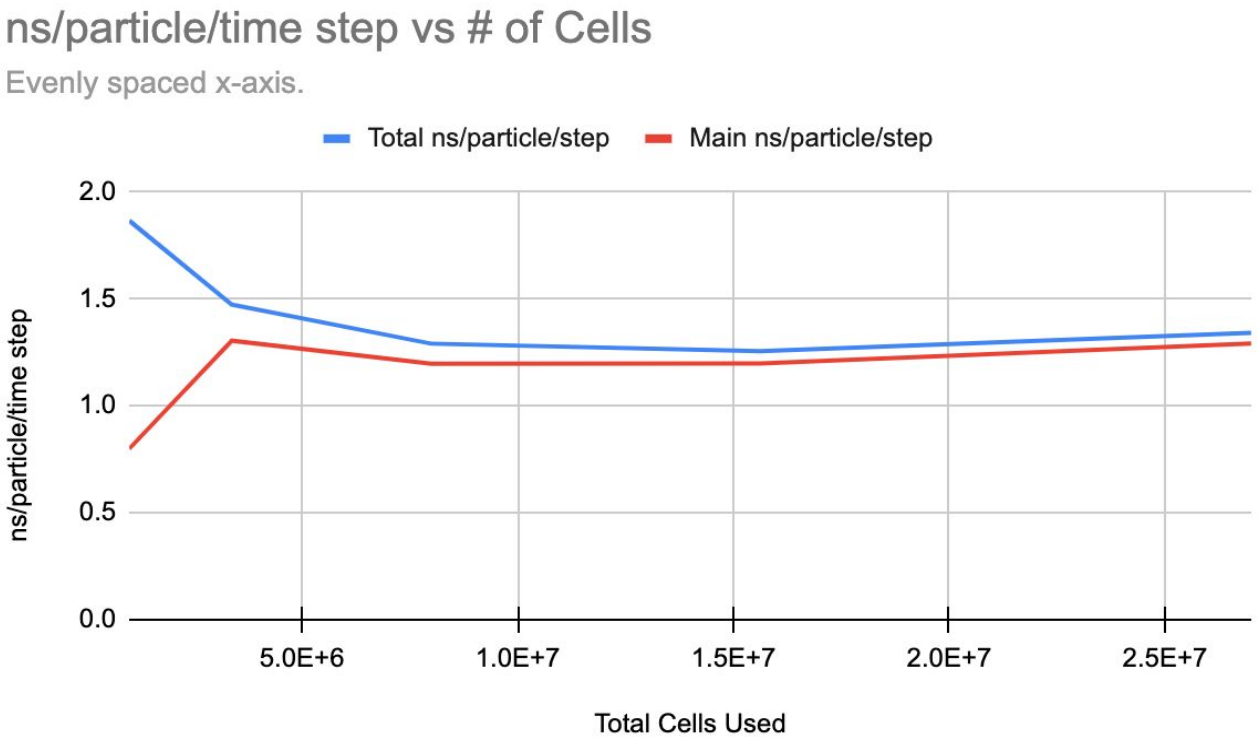
Density slices for a 300 x 300 x 300 Cuda Simulation



Other C++/Cuda Optimizations/Changes

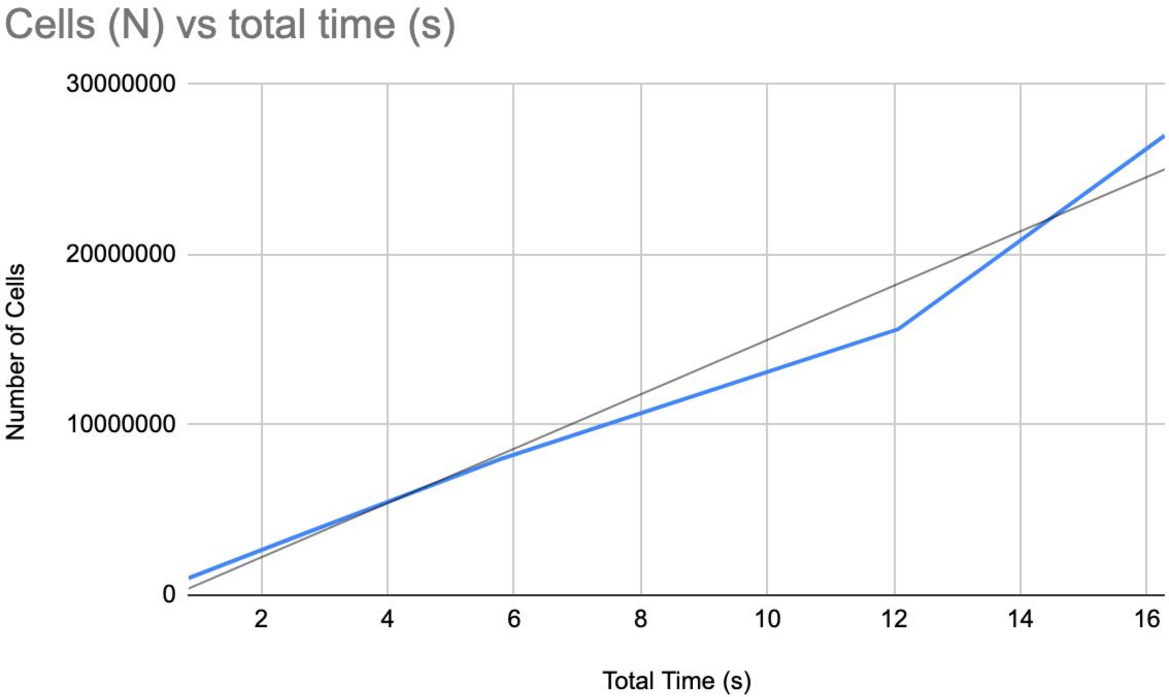
- We prefer to launch a kernel to do even simple tasks such as initializing values, resetting a grid, or calculating the new grid values
    - For a 1,000,000 cell grid to do a simple division of two numbers within arrays. Run by overhead.cu [4]
        - CPU time: 0.002s
        - CUDA Kernel time: 0.000015s
    - Cuda kernel becomes faster at a 2744 (14^3) cell grid
- Control flow statement reduction
    - Within the launched kernel, we avoid statements such as if, while, for, etc. to enable full parallelism

Current Runtime Performance



Current Runtime Performance

- Apparent convergence close to our goal of 1 ns/particle/time step
- Almost Linear N vs t
  - Small change at 150x150x150
- Initialization time barely increases
- Computation time is what is causing the runtime increases
- Hand optimizations needed



What's Next for QDS and TriForce

- We would like to utilize multiple GPU cards to perform these calculations
  - Domain decomposition being our main method
  - NVLink will also play an important role
- Use MPI with CUDA to run QDS in parallel
- Extend our Euler equations within the GPU to retrieve other important simulation values
  - This comes with finding a better way of calculating values so that threads don't interrupt each other
- A simple buffer zone that would show us how mass and energy we lost
- Incorporate QDS into the main TriForce framework PIC methods within TF-Link
  - Open up to user specifications to be compatible with TriForce

Conclusion

QDS is a new simulation method that has shown to have many advantages over our previous PIC methods, while introducing no new disadvantages. Its dynamic range appears to be limitless while simulating any sort of experiment that a user wants to take place. This has allowed for precise simulation values that a stochastic process could not replicate unless we drastically increase our particle count. Additionally, this new create-and-destroy particle cycle allows a lower memory usage while being computationally less expensive and easily improved using parallel processing and GPU computation. We are able to expand our simulations efficiently while keeping the precision needed. Overall, QDS will be able to be a new PIC method that TriForce will use to great benefit!

Questions?

For any questions or additional information, please contact me at the following email address

Email: ssikorsk@u.rochester.edu

For any questions about TriForce and what we’re working on at large, contact Dr. Adam Sefkow

Email: adam.sefkow@rochester.edu

References

[1] Albright et al. *Kinetic plasma modeling with Quiet Monte Carlo Direct Simulation*. Jan 2001.

[2] Albright et al. *Quiet direct simulation of Eulerian Fluids*. Jun 2001.

[3] Smith et al. *An improved Quiet Direct Simulation method for Eulerian fluids using a second-order scheme*. Dec 2008

[4] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[5] https://github.com/sgsikorski/3dDiffusion