New Electric Propulsion Simulation Framework for the Arduino Microcontrollers

Lubos Brieda*, Kyle Diaz† , and Rutvi Shah‡
Particle in Cell Consulting LLC, Westlake Village, CA 91362

We report on the development of a plasma simulation library for the Arduino microcontrollers. The library currently supports 1D hybrid ES-PIC simulation. Through a hardware interface, such a library could conceptually be used to actively drive an electric propulsion device during an experimental testing campaign. The library is discussed and we also discuss ongoing effort to offload some computations to an onboard FPGA.

I. Nomenclature

m	=	mass (kg)
$\vec{v}$	=	velocity (m/s)

II. Introduction

Arduino is a family of popular consumer programmable microcontrollers that is routinely used in a variety of hobbyist projects, from wearable technologies to robotics. A microcontroller is essentially a simple computer built on a single board that consists of a central processing unit, dynamic and static rewritable memory, and various input / output connections. The Arduino microcontrollers come in a variety of sizes and capabilities, as illustrated by the photograph in Figure 1. The first pictured board is the legacy Arduino Uno. The second board is the 1×2 in Arduino Nano 33 Sense BLE. This more recent board is not only smaller, but also includes a variety of integrated sensors, including a magnetometer, accelerometer, gyroscope, microphone, and a light color detector, along with a Bluetooth connectivity and a faster processor. The last board in the picture is the Arduino MKR Vidor 4000, which is the board utilized in this paper. It does not contain any of the built-in sensors of the Nano Sense, but it contains an onboard Field Programmable Gate Array (FPGA). FPGAs, as will be discussed further, are essentially programmable CPUs that allow the user to create custom instructions by specifying the path of electrical signal through logical gates. These microcontrollers run off a 5 V or a 3.3 V source, which can be provided by a battery or an USB connection. They can also be integrated with external devices, including SD card writers, LCD screens, and other sensors. The communication is accomplished using individual digital I/O pins, or I^{2}C, micro HDMI, and mini PCI express connectors. Sensors and other devices are typically shipped in the form of a breakout board with included circuitry for performing the device-specific logic as well as an Arduino software library that handles the communication. Integrating the device then simply involves electrically connecting pins from the sensor to specified pins on the Arduino board and including the appropriate library in the control software. It should be noted that Arduino is just one of several families of consumer microcontrollers. Alternatives include the Launchpad family from Texas Instruments, as well as various Arduino-compatible boards from manufacturers such as Sparkfun or Adafruit.

The small footprint and power requirements, as well as the easy connectivity to external devices makes microcontrollers somewhat similar to single board computers such as the Raspberry Pi or NVIDIA Jetson. However, these latter devices are essentially fully functional computers capable of running an operating system. It is very much feasible to install Linux or a light version of Microsoft Windows onto a Raspberry Pi, and by connecting the device to a monitor and a keyboard, one obtains a low cost desktop computer. The Arduino microcontrollers on the other hand do not offer a capability to run an an operating system. Instead, they simply allow for the use to upload a custom code to be executed continuously without requiring connectivity to a computer. As such, the Arduino is ideally suited for in-the-loop active control of hardware devices, or for real-time evaluation of collected sensor data. In fact, we have utilized an Arduino in a

*President, AIAA Senior Member, lubos.brieda@particleincell.com
†Also at Stony Brook University, Stony Brook, NY
‡Also at University of Texas, Austin, TX

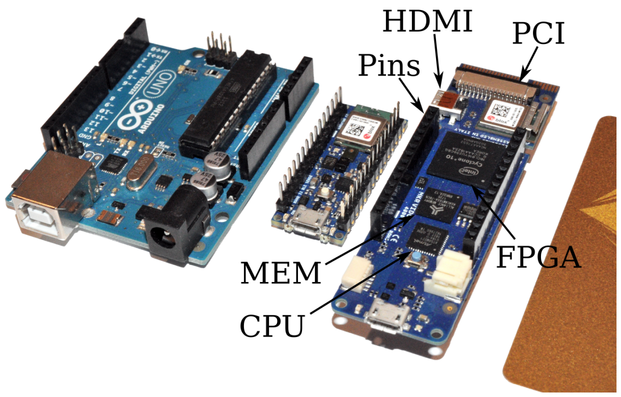


Fig. 1 Photographs of three types of Arduino boards: Uno, Nano 33 BLE Sense, and MKR Vidor 4000 with a credit card for scale

past work involving collecting environmental conditions in a gaseous purge characterization experiment[**?**]. The use of the Arduino eliminated the need for a large data acquisition board and a software license for the data collection software.

However, there are other possible uses for an Arduino beyond simple static monitoring of external sensors. It is feasible to imagine a microcontroller receiving data from external sensors, while also running a numerical simulation of the expected data. Any outlier information could then be transmitted to the user. This usage has great implications for space probes given that many science missions are bandwidth limited and require the use of compression algorithms [**?**]. In the field of spacecraft plasma propulsion, we can imagine a microcontroller using thruster operating data to tune inputs for a reduced zero or one dimensional model of the device, and then performing exploratory studies of alternate operating conditions to improve a property of interest. This could involve attempting to minimize discharge oscillations, increase exit velocities, decrease plume divergence, or increase lifetime. While the Arduino computational performance is not sufficient to simulate the device in real time, electric propulsion devices typically undergo week or month-long life test studies during which the device operates at near steady state, with variation in performance on the scale of hours or days. These time scales are sufficient for the microcontroller to establish a solution and drive the next operating set point.

For this reason, we have started the development of a plasma simulation library for the Arduino microcontrollers. The objective of this effort is to simplify deployment of plasma and rarefied gas simulation codes to the Arduino. The Arduino family was selected based on its large user base and an already existing ecosystem of peripheral devices. We start by describing the Arduino programming model, the library, and then include several examples. The performance of these codes is compared to a version running on a desktop workstation.

III. Arduino Programming

Figure 2 illustrates the actual process of programming an Arduino board. The board simply needs to be connected via a USB cable to a computer running an Arduino-provided integrated development environment (IDE). The IDE contains an editor for writing the code. It also contains a “programmer” that compiles the code and flashes it to the board over the USB connection. The IDE also contains additional useful tools such as a Serial Monitor for reading and writing data sent over the USB connection acting as a serial port. One could alternatively utilize a command line toolkit to perform the compilation and programming, and subsequently utilize custom program to perform the serial communication. The serial connection is not required once the board is programmed, however, it is useful for debugging and code monitoring. It is also how we obtain numerical results in this paper.

The Arduino is programmed in the C++ language. However, instead of implementing the `main` function, every Arduino program needs to instead implement two functions called `setup` and `loop`. The former function specifies the code that is executed whenever the board is powered up. The latter function specifies the code that subsequently runs continuously. This is analogous to a desktop code containing

```
void main() {
 setup();
 while(true) loop();
}
```

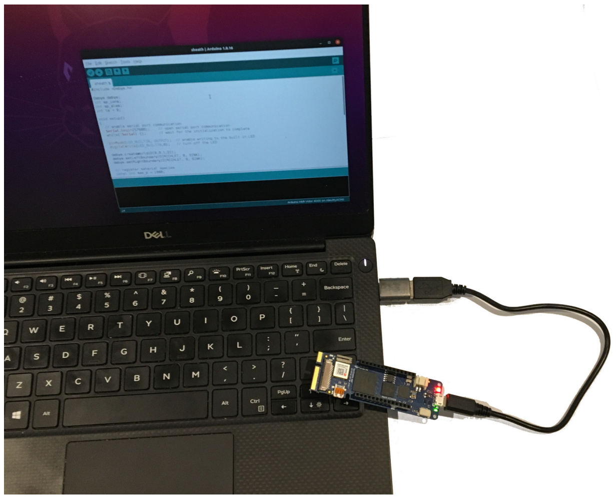


Fig. 2 Programming an Arduino is accomplished by connecting the board to a computer with a USB cable

The compiler automatically includes the `Arduino.h` header file. It defines functions for interfacing with the board. They allow us toggling digital (1/0) pins between an input and output state, and writing or reading data from digital or analog pins (analog pins are read only). Additional code can be included utilizing the standard `#include` command, with the IDE automatically linking in the corresponding library source code or precompiled binaries. One example here is the `Serial.h` library for performing communication over the serial port. Another library, `Wire.h` provides support for sending data using the I^{2}C protocol. This protocol uses two digital pins, labeled SDC and SDA, that act as a clock and a data stream. This protocol is commonly used by external peripherals and sensor breakout boards.

IV. Debye Library

Arduino also supports third-party libraries. A large ecosystem of such libraries already exists, with the code typically hosted on GitHub. Many of these libraries have been developed by the manufacturers of sensor breakout boards to simplify communication with their product. In our case, we have started the development of an Arduino plasma simulation library. The goal of this effort is to simplify the development of electric propulsion simulation codes running on the Arduino platform. The library was named Debye, given that the Debye length is the smallest spatial scale at which plasma exists, and microcontrollers are among the smallest physical devices on which a code can be run. The library is available at . The MKR Vidro 4000 board is used for testing. This board provides storage for up to 256 kB of program and static data storage, and 32 kB for dynamic variables. Current capabilities include:

- 1D ES-PIC method
- Support for multiple particle species
- Support for ionization and MCC collisions
- Direct linear and Newton Raphson non-linear Poisson Solver
- FTCS Advection-Diffusion Eulerian material
- Serial port data output, with Python script for data plotting

V. Examples

A. Ion Gun

We now illustrate the use of the library with several examples. First, let's consider a 1D simulation of a charged particle accelerator, as may be the case in ion thruster optics or in a hollow cathode. In this simple example, we assume that ions are born at the left boundary and are accelerated by an electric field established by a downstream electrode. We have $\phi(0) = 0$ and $\phi(L) = \phi_{right}$, where L is the domain length. Our goal is to compute the ion density within the device. However, perhaps due to power supply limitations, or other feedback mechanisms, let's assume that ϕ_{right} is time varying, with the value provided by a probe output. In order to approximate this connection to a physical external

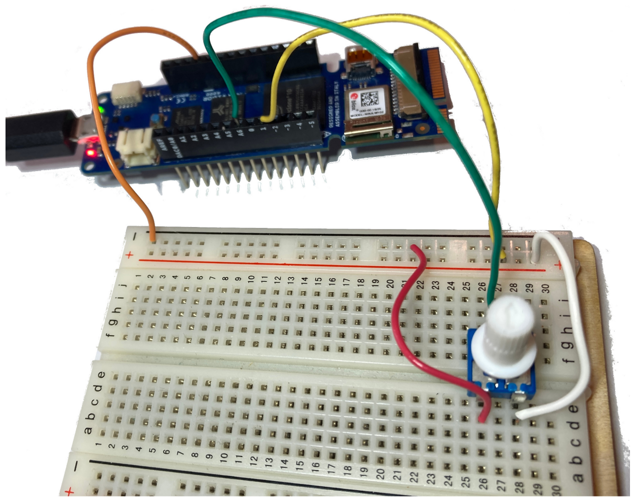


Fig. 3 Electric setup for the first example. Input voltage is provided by pin 4. The current downstream of a variable resistor is read out by analog pin 6.

envrionment, we connect the microcontroller to a variable resistor, as illustrated in Figure 3. Input voltage to the circuit is provided by pin 4. The resistor is then connected in series. The output is connected to the Arduino analog pin 6. The current flowing into this pin can be readout using Arduino's `analogRead` function, which returns an integern value in range [0 : 1023].

The ion flow can be simulated using the kinetic Particle in Cell (PIC) method. In the electrostatic implementation, we use the Poisson's equation

$$\nabla^2\phi = -\rho/\epsilon_0 \tag{1}$$

to compute plasma potential ϕ given some charge density ρ, where the permittivity of free space ϵ_0 is a physical constant. Plasma potential is related to the electric field $\vec{E}$ via

$$\vec{E} = -\nabla\phi \tag{2}$$

The electric field is subsequently used to advance ion velocities and positions by integrating the equations of motion

$$d\vec{v}/dt = q\vec{E}/m \tag{3}$$

$$d\vec{x}/dt = \vec{v} \tag{4}$$

through small Δt time steps. The force term in Equation 3 ignores the magnetic field term, which is not present in our setup. A volumetric grid is used to compute ρ from individual particle positions.

Setting up this simulation with the Debye library is trivial. We begin by including the `<Debye.h>` header file and instantiating a global `Debye` object. We then declare various simulation parameters. These include values for the domain extents, $x \in [x_0, x_m)$, the number of volume mesh nodes, and the Δt simulation time step size. We also specify the maximum number of simulation particles, and declare variables to hold the ion material index and the current time step.

```
#include <Debye.h>
Debye debye;                    // library instance

// declare simulation parameters
const float x0 = 0.0;          // mesh origin
const float xm = 0.1;          // mesh extend
const int ni = 51;             // number of nodes
const float dx = (xm-x0)/(ni-1);   // cell spacing
const float dt = 5e-8;         // time step

const int max_p = 2000;        // maximum number of simulation particles

// global variables
int mat_ions;                  // material index for ions
int ts = 0;                    // current time step
```

1. Setup function

We next populate the `setup` function that runs on the initial power up. It begins by establishing serial communication to be used for a diagnostic output. In a production run in which the microcontroller is run independently of a computer, this step would be omitted. We also use Arduino standard library functions to toggle the digital pin connected to an onboard LED into the output (write) mode. Subsequently we use `digitalWrite` to set this pin to the low state to turn off the LED. The LED will be turned on later on to indicate start up progress. We also use signal from analog pin 0 to initialize the random number generator. This pin is not electrically connected and as such, will read out a random value.

```
// code to run on initial board power up
void setup() {
  // enable serial port communication
  Serial.begin(57600);      // open serial port communication
  while(!Serial) {};        // wait for the initialization to complete

  pinMode(LED_BUILTIN, OUTPUT);  // enable writing to the built in LED
  digitalWrite(LED_BUILTIN,0);   // turn off the LED

  // initialize RNG
  randomSeed(analogRead(0));
```

Subsequently, we use the `createWorld1D` function to initialize the computational domain. This function allocates memory for a 1D Cartesian grid spanning the given extents, and containing the given number of computational nodes. We then specify the left and right boundary conditions. In this case, both are set to a fixed (Dirichlet) potential. The boundaries are initialized to $\phi_{left} = 0$ V and $\phi_{right} = -2$ V.

```
  debye.createWorld1D(x0,xm,ni);
  debye.setLeftBoundary(DIRICHLET, 0);
  debye.setRightBoundary(DIRICHLET, -2);
```

Next, we use the `registerParticleMat` member function to initialize a new material species that is represented using simulation particles. The physical mass and charge of the physical ions is also specified. The final argument specifies the maximum number of particles which controls the size of the allocated memory structure.

```
  // register material species
  mat_ions = debye.registerParticleMat(16*Const::AMU,1*Const::QE,max_p);
```

We next make sure that the library initialization succeeded. The `Debye` class overrides the Boolean truth operator, allowing us to check error status in a manner similar to other Arduino classes. We indicate success by blinking the onboard LED 4 times. Otherwise, we "hang" the board by jumping into an infinite `while(true) ...` loop.

```
  // make sure it all went well
  if (debye) {
        for (int i=0;i<4;i++) {                    // blink the LED 4x to update status
        digitalWrite(LED_BUILTIN,1); delay(200);
        digitalWrite(LED_BUILTIN,0); delay(200);
        }
  }
  else {Serial.println("Error␣starting␣up␣Debye!");
        while(true) {delay(1000); } // block execution
}
```

Finally, we active the external controller circuit by setting digital pin to the high state. This operation effectively creates a 5V (on the MKR Vidor board) voltage drop across the external circuit.

```
  // activate controller
  pinMode(1, OUTPUT);
  digitalWrite(1,HIGH);
}
```

2. Loop function

Next, we populate the `loop` function. We begin by reading a value from the analog pin 6. The integer value is converted to a real number in range $[-100, 100)$, which is used to update potential on the right boundary ϕ_{right}.

```
// code to run indefinitely
void loop() {
```

```
int val = analogRead(6);
float f = val/1023.0;      // convert to [0:1)
debye.setRightBoundary(DIRICHLET, 100*(-1+2*f)); // set phi in [-100,100)
```

We then use `loadParticles` member function to inject 5 particles at random position $x \in [x_0, x_0 + 5\Delta x)$. The particles are born with zero drift and thermal velocity corresponding to 0.1 eV. Their macroparticle weight is set such that these particles correspond to n_0 number density, $w_{mp} = n_0\,(5\Delta x \cdot 1 \cdot 1)\,/5$.

```
// inject thermal ions in the first 5 cells
debye.loadParticles(mat_ions,x0,x0+5*dx,nd0,0.1*Const::EvToK,5);
```

Next we call the `advance` member function. This function implements the actual PIC algorithm. It uses particle positions to compute charge density, uses it to compute plasma potential and the electric field, and finally integrates particle velocities and position through the given time step.

```
debye.advance(dt);   // computes electric field and advances particles
```

Finally, we increment a local variable `ts` that keeps track of the current time step. Every 25 steps, we call the `serialWrite` function to output simulation data to the serial port. We also toggle the board built-in LED to provide visual feedback to the user.

```
 ts++;
 if (ts%25) {
     debye.serialWrite();
     digitalWrite(LED_BUILTIN,!digitalRead(LED_BUILTIN));
 }
}
```

3. Diagnostic Output

The previously mentioned `serialWrite` function outputs information on the state of the simulation to the serial port in the form of a comma-separated text line. The stream consists of the following entries:

```
ts,num_mats,np0,np1,...,np_(num_mats-1),ni,phi[0],phi[1],...,phi[ni-1],
mat[0].nd[0],...,mat[0].nd[ni-1],mat[1].nd[0],...,mat[num_mats-1].nd[ni-1]
```

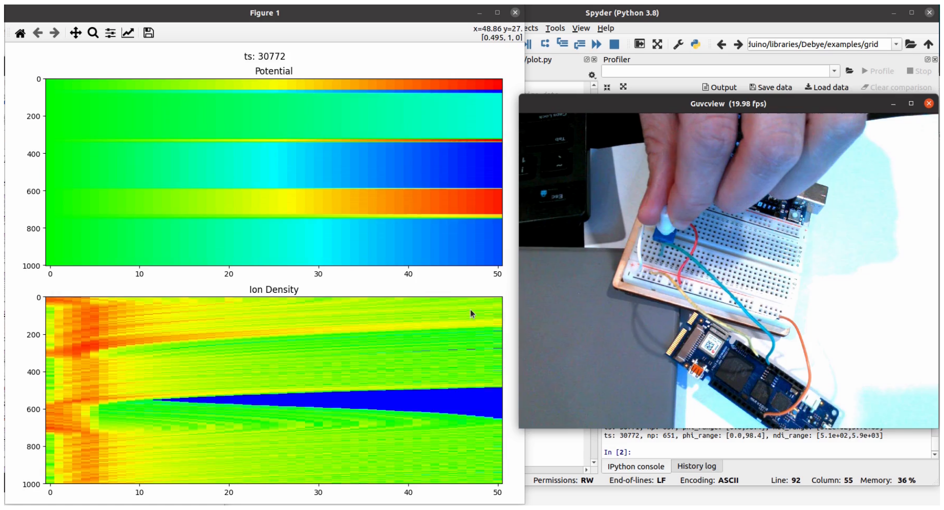
Additional output can be achieved by modifying the source code in `Debye.cpp` or by implementing a custom function. While the Arduino IDE offers a serial plotter, the plotter is limited to visualizing line plots with a single y-axis. We instead use a Python script to read and plot the data. The script uses Python `serial` library to read data from the serial port. It begins by searching for an available port,

```
for i in range(0,5):
    try:
        port_name = "/dev/ttyACM%d"%i
        ser = serial.Serial(port_name,57600)
        print("Opened␣port␣"+port_name)
        port_ok = True
        break
    except serial.SerialException:
        pass
```

Subsequently, we use `matplotlib.animation.FuncAnimation` to continuously call `anim` function, which begins by reading a line from the serial port. This function blocks until the line available. The data read from the Arduino is encoded as a string literal, and the second line strips out the relevant info. Alternatively, we could use `line.decode('utf-8').rstrip()` to accomplish the same. Finally, the data is split by comma to obtain a list of individual entries.

```
def anim(n):
    line = str(ser.readline()); # read line from serial port
    line = line[2:-5]   #eliminate trailing b' and \r\n
    pieces = line.split(','); # split by comma
```

For brevity, the full source code of the plotting function is not included here, but is available in the Examples folder provided with the Debye library. The data is however read out using syntax such as



```
ts: 30772, np: 651, phi_range: [0.0,98.4], ndi_range: [5.1e+02,5.9e+03]

In [2]:
```

Fig. 4 Example of real-time feedback to user input

```
ts = int(pieces.pop(0))
...
for i in range(ni):
    phiI[i] = float(pieces.pop(0))
```

The collected values are used to update data within an `imshow` object to produce a waterfall time history plot, as shown in Figure 4. The time increases from top to bottom. The top plot shows the plasma potential while the bottom plot is the ion number density. As expected, by increasing the potential on the downstream electrode (indicated by the red coloring), the ions are pushed away from this boundary due to a formation of a retarding electric field. This is indicated by both the lack of ions away from the injection region (the blue coloring in the number density plot), and the upstream migration of the high ion density region (the red region near the left boundary).

B. Plasma Sheath

The above example simulated only the ion population. We next demonstrate the use of the library to run a fully-kinetic simulation of a plasma sheath. For this simulation we again consider a one-dimensional domain spanning $x \in [0, 0.1)$ m divided into 30 Cartesian cell. Potential on the left and right boundary is fixed to 0V.

```
// declare simulation parameters
const float x0 = 0.0;        // mesh origin
const float xm = 0.1;        // mesh extend
const int ni = 31;           // number of nodes
const float dx = xm/(ni-1);  // cell spacing
const float dt = 1e-9;       // time step

const float nd0 = 1e12;      // reference number density
const float phi0 = 0;        // reference potential
const float kTe0 = 1;        // reference electron temperature

const int max_p = 1200;      // maximum number of simulation particles

void setup() {
  ...
  debye.createWorld1D(x0,xm,ni);
  debye.setLeftBoundary(DIRICHLET, 0);
  debye.setRightBoundary(DIRICHLET, 0);
```

We next use `registerParticleMat` to initialize material species corresponding to H^+ protons and e^- electrons. We then use use `loadParticles` to load `max_p` randomly positioned particles of each species throughout the computational domain.

```
  // register material species
  mat_ions = debye.registerParticleMat(1*Const::AMU,1*Const::QE,max_p);
  mat_eles = debye.registerParticleMat(Const::ME,-1*Const::QE,max_p);

  debye.loadParticles(mat_ions,x0,xm,nd0,0.1*Const::EvToK,max_p);
  debye.loadParticles(mat_eles,x0,xm,nd0,1*Const::EvToK,max_p);
  ...
}
```

The `loop` function essentially consists of calling `advance`. Every 25 time steps we write out results to the serial port using `serialWrite`.

```
void loop() {
 debye.advance(dt);   // computes electric field and advances particles

 ts++;
 if (ts%25==0) {
        debye.serialWrite();
 }
}
```

Figure 5 plasma potential, charge density, and ion and electron number densities after 422 and 6549 time steps (these are arbitrary time points at which the screenshot was grabbed). In the former plot, we can observe an essentially uniform ion density, while electrons show depletion (dark brown/gray color). The charge density also shows the net positive charge (red coloring) in the charge density plot. At the later time step, ion depletion near the walls is now apparent, and the electron population is decreased even further. However, charge imbalance in the near-wall sheath is decreased, as expected. The oscillations in the plasma potential profile arise from the plasma attempting to retain charge neutrality in the bulk region. The magnitude is related to numerical noise, which could be reduced by using more simulation particles. However, the reduced available memory space on the Arduino does not allow us to use a significantly larger number of particles.

Since Arduino is programmed essentially in C++, it is straightforward to migrate an Arduino code to a desktop. This was done for this example, with the Debye library source files modified to remove calls to Arduino-specific functions (such as for performing serial port communication or flashing the LED). This conversion allows us to compare the performance of the Arduino to a typical computer run time. Using the MKR Vidor 4000 board, we can simulate 500 time steps with about 2000 total particles and 30 computational cells and data output once every 100 time steps, in 164 seconds (2 minutes, 44 seconds). The same configuration requires 0.0149 seconds on an Intel i5-8265 1.60 GHz CPU laptop. The laptop timing includes a screen output at the same 100 time step frequency as the Arduino board. The Arduino simulation thus runs 1100× slower than the desktop. While this may seem like a significant difference, not even the laptop computer is capable of running at real-times.

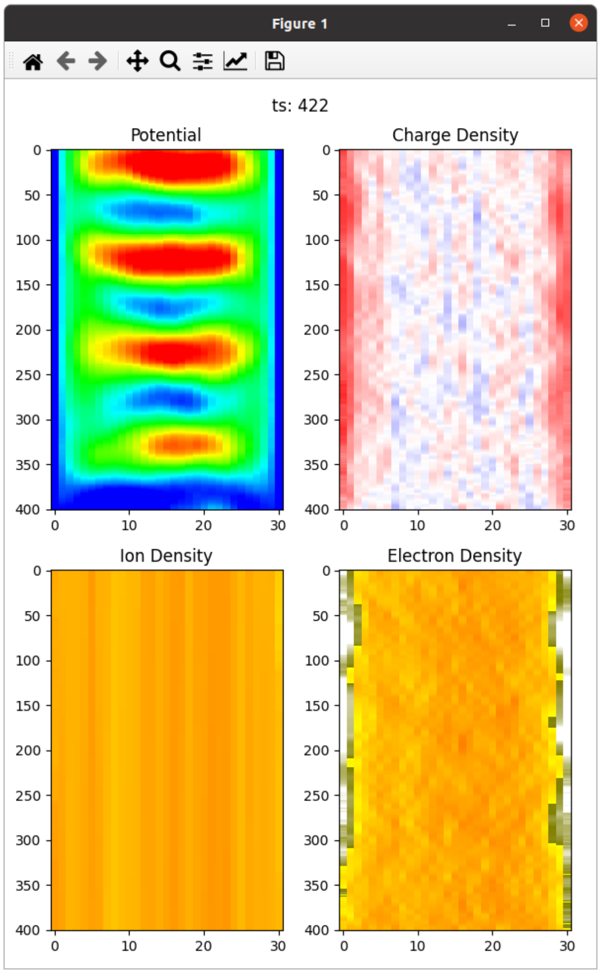
VI. Future Work

A. Additional Material Types

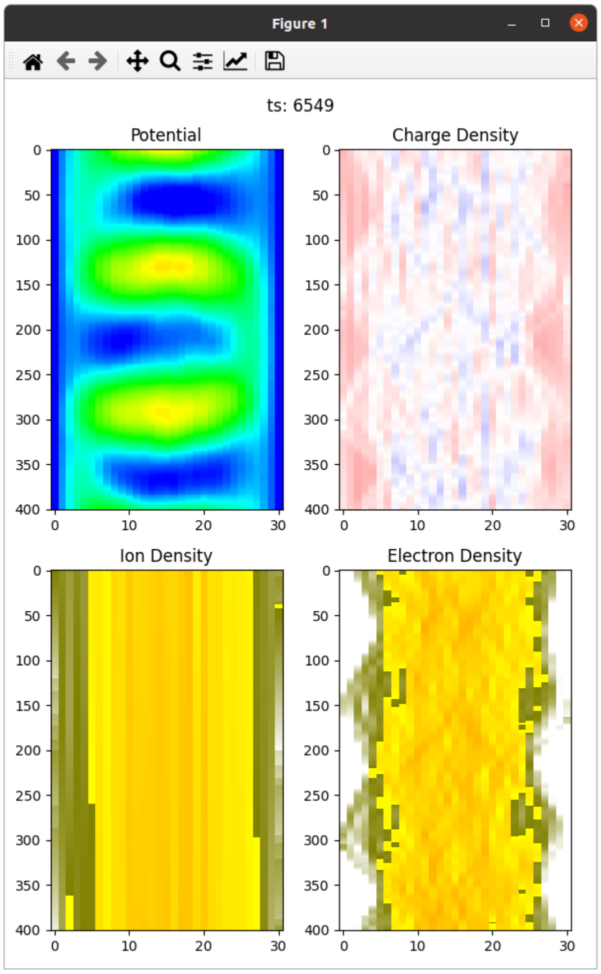
While the basics of the library have been developed, much work remains to turn the library into a useful product. Given the desired applicability to electric propulsion, our immediate goal is to develop additional material models to simplify modeling of EP devices. As an example, we have developed a stand-alone 1D model of a Hall thruster based on the thermalized model of Fife. This model is currently being ported to the Arduino. T

B. FPGA Interoperability

MKR Vidor 4000 selected due to the presence of an onboard Field Programmable Gate Array (FPGA) FPGA are essentially a large array of logical electrical gates (multiplexers). As an example, an "AND" gate passes high signal if both inputs are high. Programmable lookup table (LUT) elements specify the sequence of active gates. The primary benefit of the FPGA is that essentially allows one to create custom "CPU" instructions by programatically setting the path of electricity through the device. FPGA prices vary according to the number of logical elements, among other things. The budget Intel Cyclone 10LP board included with the Arduino MKR Vidor board contains 15408 logical elements. This is a sufficient count for creating a handful of custom instructions, especially if limited to elementary fixed-point math. The use of floating point mathematics and trigonometric or exponential functions requires including "intellectual property" (IP) blocks implementing these arithmetic operators. While implementation specific, in [**?**] the



(a) 422



(b) 6549

Fig. 5 Plasma sheath simulation results after 422 and 6549 time steps

authors report 1907 logical elements required to implement a square root operator using an algorithm of Goldschmidt, while 3150 LUTs are needed to implement an add or subtract operation on 64-bit floating point values. For a production level FPGA, these LUT counts represent a minor fraction of the available resources, however, given the budget Intel Cyclone 10LP board on the Arduino MKR Vidor with only 15408 gates, we need to be careful with the use of resources.

VII. Conclusion

We have developed and demonstrated a new method for modeling the dynamic evolution of surface in contact with plasma based on coupling the PIC and DEM methods. The surface layer is represented by spherical elements interacting with each other via spring forces, up to some maximum threshold. Ion collision with grain impart additional momentum to the grains. This approach allows us to dynamically evolve the surface roughness and material compositions of a surface layer at spatial and temporal scales greater than achievable with MD. We use the model, along with a simple scheme for charge propagation, and a Maxwellian beam approximation of a cathode spot ion emission, to simulate the dynamic evolution of conductivity across an insulator layer. We observe numerically, that similarly to what is seen in the experiment, a conductive bridge forms that is speculated to have implications on the thruster life time. As part of future work, we plan to develop a more physically sound model for the cathode spot, include chemical phase change to model condensation, and include a Poisson solver to simulate electrostatic effects. Nevertheless, this combined PIC-DEM approach appears to be an attractive scheme for modeling the near-surface plasma domain.

Appendix

The source code and animations videos are available at `https://www.particleincell.com/2021/xx`.