

Starfish Webinar

January 25th, 2018

Lubos Brieda, Ph.D.
Particle In Cell Consulting LLC
particleincell.com/starfish



Bio

- **Speaker Bio: Lubos Brieda.**
 - PhD in Mechanical and Aerospace Engineering from George Washington University in 2012
 - Advisor: Prof. Michael Keidar
 - Thesis topic: multiscale modeling of Hall thrusters
 - Master's Degree in Aerospace Engineering from Virginia Tech in 2005
 - Advisor: Prof. Joe Wang
 - Thesis topic: fully kinetic simulations of ion beam neutralization with a 3D code
 - Work experience:
 - Air Force Research Lab 2005-2008
 - NASA Goddard 2008-2012
 - Particle In Cell Consulting 2008 – present
 - Research Interests:
 - Plasma modeling
 - Electric (plasma) propulsion
 - Contamination transport

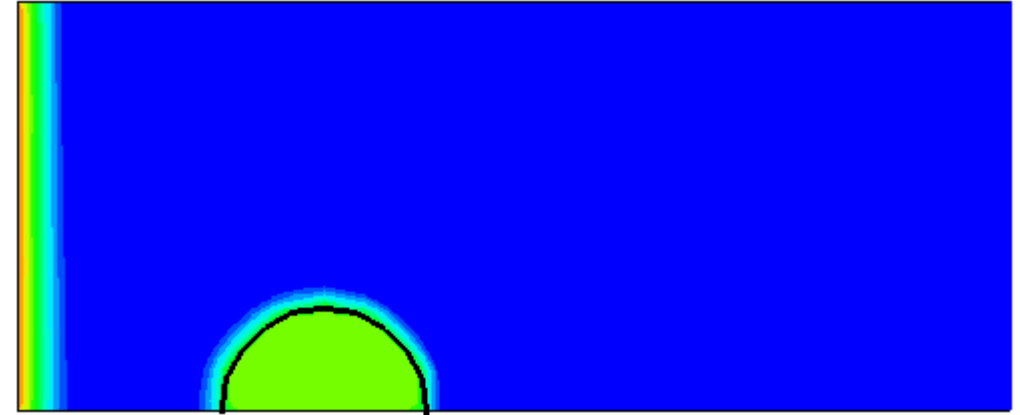
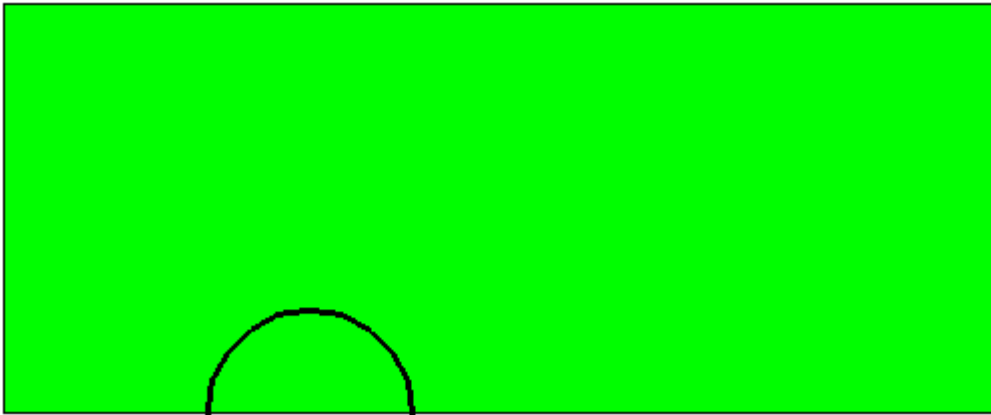


Courses

- I also offer online courses on various aspects of plasma modeling:
 - [Fundamentals of the PIC method](#): This course introduces the Particle in Cell method used for kinetic plasma simulations using a step-by-step approach. We will develop 1D, 3D, and 2D (axisymmetric) codes to simulate plasma sheath, $E \times B$ transport, plasma flow past a charged sphere, and a simple ion gun.
 - [Advanced PIC techniques](#): This course covers topics beyond the scope of the intro course. It covers three main concepts: electromagnetic PIC (EM-PIC), Direct Simulation Monte Carlo (DSMC) collision modeling, and finite element PIC (FEM-PIC).
 - [Distributed Computing for Plasma Simulations](#): In this course you'll learn how to develop plasma simulation codes that utilize multiple CPUs and graphic cards to handle larger simulation domains or to run faster. We'll cover multithreading, distributed computing with MPI, and GPU computing using CUDA.
 - [Fluid modeling of plasmas](#) (March 2018): This new course will teach you how simulate dense plasmas in which the continuum assumption holds. We will cover single and multi-fluid MHD equations as well as hybrid approaches with detailed electron model and some advanced topics like Vlasov solvers.
- Please see <https://www.particleincell.com/courses/> for more info and to sign up
 - Early bird rate for the fluid modeling course ends February 6th

About Starfish

- What is Starfish?
 - Two-Dimensional (XY or RZ) Java code for modeling ionized and non-ionized gases
 - There are two editions: “regular” and “light”
 - The light edition, Starfish-LE, meant to be an academic tool that you can use to learn about modeling plasmas and rarefied gases and possibly extend with your own models <- **Topic of this webinar**
 - You can download the binary from <https://www.particleincell.com/starfish/> and get the source code from <https://github.com/particleincell/Starfish-LE>
 - There you will also find links to a five step tutorial for getting started as an end user
 - Today we will briefly review the tutorials and then review development within Starfish



Starfish Features

- **Computational Domain:** 2D XY and RZ support utilizing one or more rectilinear meshes
- **Surface Geometry:** defined using linear and/or cubic splines specified in SVG-like format.
- **Materials:** multiple fluid and/or kinetic species
- **Material Interactions:** DSMC (particle-particle), MCC (particle-fluid), and chemical reactions (fluid-fluid)
- **Sources:** surface and volume sources such as Maxwellian, ambient pressure,
- **Output:** field, surface, and particle data saved in Tecplot or Paraview format, support for animation and data averaging
- **Solvers (Starfish-LE):** electrostatic particle in cell (ES-PIC), diffusion equation solver
- **Parallelization:** multithreaded particle push
- **Short term wish list:** GUI, better parallel processing, electromagnetic model (EM-PIC), MHD model for plasma, adaptive mesh refinement

Starfish Continued

- Starfish is a command line code
 - GUI development right now on hold
- Commands specified in an XML starfish.xml file
- Utilizing standard XML syntax, the file consists of numerous **<command> ... </command>** elements
- Each command element can contain multiple **attributes** or **child nodes**

<command attr="value">

<node1>node1_value</node1>

<node2>node2_value</node2>

</command>

- There is no difference in using attributes or nodes to specify value
- Inputs can be split into multiple files loaded with a **<load>** command

```
<simulation>
<note>Starfish Tutorial: Part 1</note>

<!-- load input files -->
<load>domain.xml</load>
<load>materials.xml</load>
<load>cylinder.xml</load>

<!-- set potential solver -->
<solver type="poisson">
  <n0>1e12</n0>
  <Te0>1.5</Te0>
  <phi0>0</phi0>
  <max_it>1e4</max_it>
  <nl_max_it>25</nl_max_it>
  <tol>1e-4</tol>
  <nl_tol>1e-3</nl_tol>
  <linear>false</linear>
</solver>

<!-- set time parameters -->
<time>
  <num_it>0</num_it>
  <dt>5e-7</dt>
</time>

<!-- run simulation -->
<starfish />

<!-- save results -->
<output type="2D" file_name="field.dat" format="tecplot">
  <variables>phi, efi, efj, rho, nd.o+</variables>
</output>

<output type="1D" file_name="profile.dat" format="tecplot">
  <mesh>mesh1</mesh>
  <index>J=0</index>
  <variables>phi, efi, efj, rho, nd.o+</variables>
</output>

<output type="boundaries" file_name="boundaries.dat" format="tecplot" />

</simulation>
```

Starfish Continued

- Without getting into much programming detail (yet), Starfish was developed as a flexible, easily extensible framework
- It utilize concepts from Object Oriented Programming
- As an example, consider a general gas **material**. In order to perform a gas simulation, we need densities, temperatures, a bulk velocities for all gas materials present in the simulation at some time step **k**.
- There are many methods that can be used to update these properties depending on problem details:
 - Particle in Cell (PIC) method for low density plasmas
 - Navier Stokes (NS) solver for dense neutral gases
 - Magnetohydrodynamics (MHD) for dense plasmas, etc...
- Using OOP, we can define a generic concept of a material that can somehow integrate its properties to a new time step
- The main simulation driver no longer needs to care about what numerical method is used

Basic Algorithm

- The code implements the following algorithm

```
initialize command modules based on starfish.xml  
while (time < max_time):  
    update global fields (plasma potential, etc...)  
    sample sources (injects new materials)  
    update materials (integrates densities, velocities ...)  
    perform interactions (inter-material collisions or chemical reactions)  
    save restart data (optional)  
    animation save (optional, output of visualization files)  
    averaging sample (optional, averaging to smooth out results)  
    print stats (writes information to the screen and log file)  
    time advance (advances simulation time)  
finalize command modules
```

- The user specified information in **starfish.xml** defines the actual algorithms performed within each of the above steps

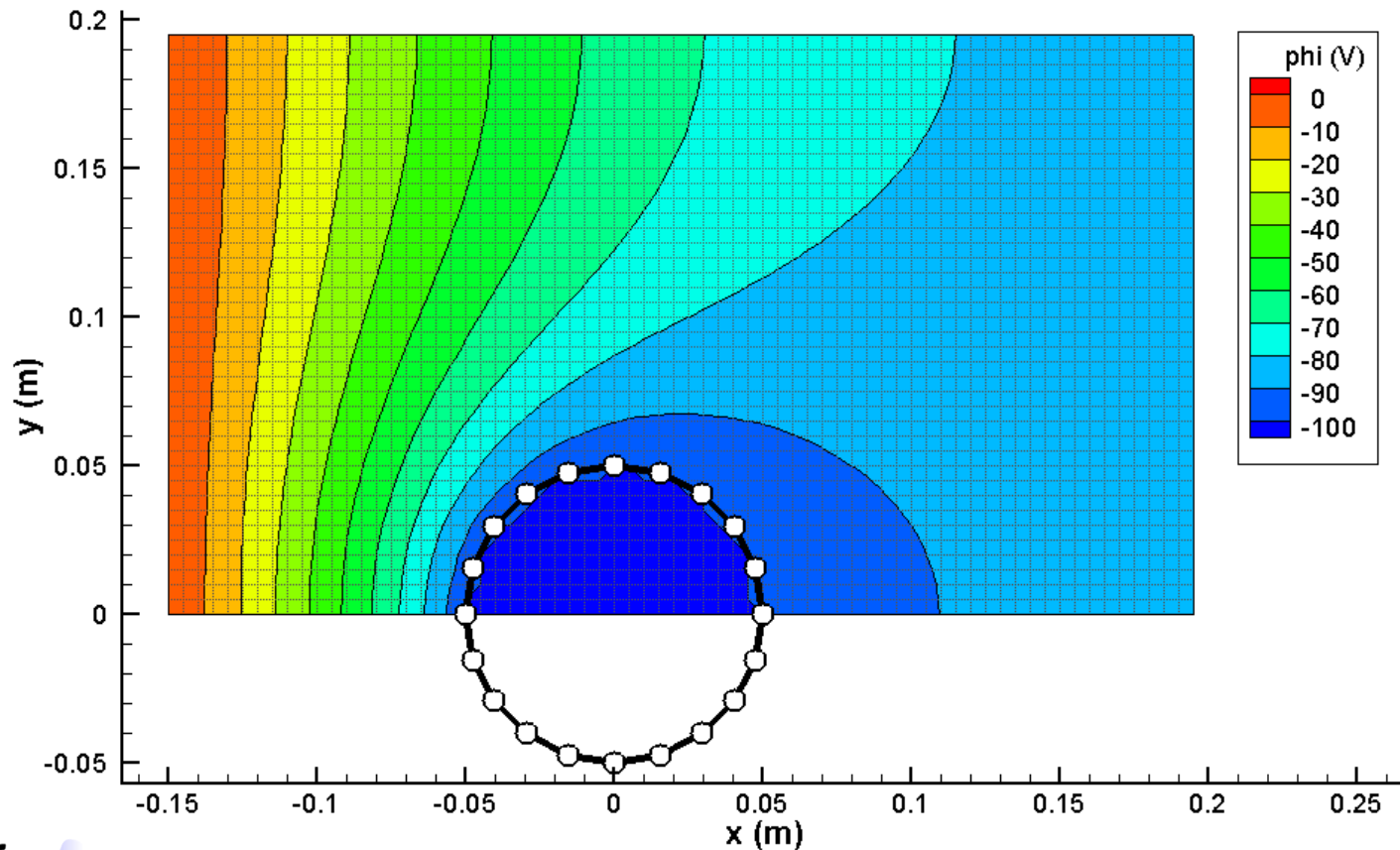
Modules

- Here is the list of currently defined command modules in the order of appearance

Module	Purpose	Module	Purpose
note	Adds user defined message to the log file	time	Controls time step and code termination
boundaries	Loads line segments defining surface geometry and contains math functions for line-line intersections	load_field	Support for loading magnetic (and other) fields
domain	Generates computational mesh(es)	restart	Support for restarting simulation
materials	Loads definition of solid or gas materials	stop	Terminates the code (useful for debugging)
material_interactions	Handles surface interactions, chemistry, and collisions	starfish	Provides simulation main loop
sources	Contains mass injection algorithms	particle_trace	Output of a single particle to a file
solver	Various field solvers (such as Poisson)	animation	Generates output files at user defined interval
output	Functions for generating output files	averaging	Data averaging

Flow Past a Sphere

- We will go through the steps for setting up a simple simulation of ions flowing past a charged sphere, from <https://www.particleincell.com/2012/starfish-tutorial-part1/>
- We start by defining problem geometry and solving the initial field



```
<simulation>
<note>Starfish Tutorial: Part 1</note>

<!-- load input files -->
<load>domain.xml</load>
<load>materials.xml</load>
<load>cylinder.xml</load>

<!-- set potential solver -->
<solver type="poisson">
  <n0>1e12</n0>
  <Te0>1.5</Te0>
  <phi0>0</phi0>
  <max_it>1e4</max_it>
  <nl_max_it>25</nl_max_it>
  <tol>1e-4</tol>
  <nl_tol>1e-3</nl_tol>
  <linear>false</linear>
</solver>

<!-- set time parameters -->
<time>
  <num_it>0</num_it>
  <dt>5e-7</dt>
</time>

<!-- run simulation -->
<starfish />

<!-- save results -->
<output type="2D" file_name="field.dat" format="tecplot">
  <variables>phi, efi, efj, rho, nd.O+</variables>
</output>

<output type="boundaries" file_name="boundaries.dat"
format="tecplot" />

</simulation>
```

Domain

- The **<domain>** command (including from domain.xml) specifies details of the **computational domain**

- This is the computational mesh used to compute gas density or solve plasma potential

- Syntax is

<domain>

<mesh> ... </mesh>

<mesh> ... </mesh>

</domain>

- Each **<mesh>** child then specifies additional parameters such as type (uniform), origin, spacing, and number of nodes
- Mesh boundary conditions **<mesh-bc>** can also be specified. Open boundary is the default.

```
<domain type="xy">  
  
  <mesh type="uniform" name="mesh1">  
    <origin>-0.15,0</origin>  
    <spacing>5e-3, 5e-3</spacing>  
    <nodes>70, 40</nodes>  
    <mesh-bc wall="left" type="dirichlet" value="0" />  
  </mesh>  
</domain>
```

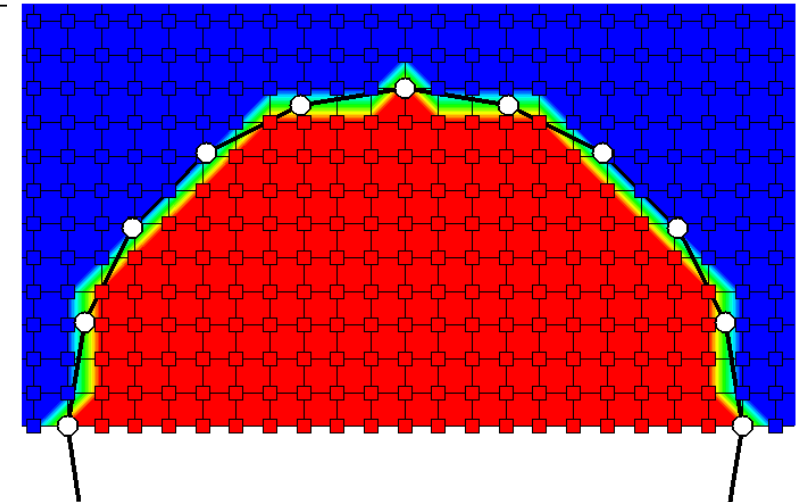
Boundaries

- **<boundaries>** command (cylinder.xml) specifies surface geometry using an SVG-like path
- Multiple boundaries can be listed
- Boundaries can be joined together using **name:first** or **name:last**
 - `<path>M cylinder:last L 2 1 ...</path>`
- Very important: the code uses right hand orientation to set normal vectors
- Normals need to face into the gas domain, the region opposite the normal is the solid domain (with the code using sugarcubing)
- Incorrect normals orientation will result in the code treating the gas region as the solid domain

```
<boundaries>

<boundary name="cylinder" type="solid" value="-100"
reverse="false">
<material>SS</material>
<path>M 0.05, 0 L 0.0475528, -0.0154508 0.0404508, -
0.0293893 0.0293893, -0.0404508 0.0154508, -0.0475528 -
9.18E-18, -0.05 -0.0154508, -0.0475528 -0.0293893, -
0.0404508 -0.0404508, -0.0293893 -0.0475528, -0.0154508
-0.05, 6.12E-18 -0.0475528, 0.0154508 -0.0404508,
0.0293893 -0.0293893, 0.0404508 -0.0154508, 0.0475528
3.06E-18, 0.05 0.0154508, 0.0475528 0.0293893,
0.0404508 0.0404508, 0.0293893 0.0475528, 0.0154508
0.05, 0</path>
</boundary>

</boundaries>
```



Materials

- The **<materials>** command specifies different materials
- Material type controls how the density and velocity is updated
 - **solid** materials are time invariant
 - **kinetic** materials use the particle in cell method
 - **fluid_diffusion** solves the diffusion equation (note, this material is temporarily removed from Starfish-LE but will be reintegrated shortly)
- You can register new material types by developing solver plugins

```
<!-- materials file -->
<materials>

<material name="O+" type="kinetic">
  <molwt>16</molwt>
  <charge>1</charge>
  <spwt>5e9</spwt>
  <init>nd_back=1e4</init>
</material>

<material name="SS" type="solid">
  <molwt>52.3</molwt>
  <density>8000</density>
</material>

</materials>
```

Solver

- Field solvers are specified with a **<solver>** command

- Here we specify a non-linear Poisson solver that solves

$$\nabla^2 \phi = -e \left[n_i - n_0 \exp \left(\frac{\phi - \phi_0}{kT_e} \right) \right]$$

- We also specify number of simulation time steps (zero to just get the initial field) with **<time>** and then run the simulation with **<starfish />**
- Results are saved in the Tecplot format using

```
<!-- save results -->
<output type="2D" file_name="field.dat" format="tecplot">
<variables>phi, efi, efj, rho, nd.O+</variables>
</output>

<output type="1D" file_name="profile.dat" format="tecplot">
<mesh>mesh1</mesh>
<index>J=0</index>
<variables>phi, efi, efj, rho, nd.o+</variables>
</output>
```

```
<!-- set potential solver -->
<solver type="poisson">
<n0>1e12</n0>
<Te0>1.5</Te0>
<phi0>0</phi0>
<max_it>1e4</max_it>
<nl_max_it>25</nl_max_it>
<tol>1e-4</tol>
<nl_tol>1e-3</nl_tol>
<linear>false</linear>
</solver>
```

```
<!-- set time parameters -->
<time>
<num_it>0</num_it>
<dt>5e-7</dt>
</time>
```

```
<!-- run simulation -->
<starfish />
```

- You should see similar output when running the code

```
C:\codes\starfish\dat\tutorial\step1>java -jar starfish.jar
=====
> Starfish v0.17 LE (Development)
> General 2D Plasma / Gas Kinetic Code
> (c) 2012-2017, Particle In Cell Consulting LLC
> info@particleincell.com, www.particleincell.com

!! This is a development version. The software is provided as-is,
!! with no implied or expressed warranties. Report bugs to
!! bugs@particleincell.com
=====

Processing <note>
**Starfish Tutorial: Part 1**
Processing <domain>
Processing <materials>
Processing <boundaries>
Processing <solver>
Processing <time>
Processing <starfish>
Starting main loop
it: 0    O2+: 0
WARNING:  !! GS failed to converge in 10000 iteration, norm = 1.3797628795224026
WARNING:  !! GS failed to converge in 10000 iteration, norm = 0.004582671221260977
WARNING:  !! GS failed to converge in 10000 iteration, norm = 6.91735940453368E-6
Processing <output>
Processing <output>
Processing <output>
Done!
```

Adding particles and interactions

- In tutorial step 3, we add interparticle interactions and also specify a surface source to inject particles
- We need a new boundary spline to associate with the source (in boundaries.xml)
 - This is a straight line along the left domain face with the normal pointing in the +X direction

```
<boundary name="inlet" type="virtual" >
<path>M -0.15,0.2 L -0.15, 0</path>
</boundary>
```

- We use a **uniform** source to inject particles. This source generates a cold beam with constant \dot{m} in kg/s

```
<simulation>
<note>Starfish Tutorial: Part 3</note>

<!-- load input files -->
<load>domain.xml</load>
<load>materials.xml</load>
<load>boundaries.xml</load>
<load>interactions.xml</load>

<!-- set sources -->
<sources>

<boundary_source name="space">
<type>uniform</type>
<material>O+</material>
<boundary>inlet</boundary>
<mdot>5.313e-11</mdot>
<v_drift>10000</v_drift>
</boundary_source>

</sources>

<!-- set time parameters -->
<time>
<num_it>500</num_it>
<dt>5e-7</dt>
</time>
...
</simulation>
```


Material Interactions

- **<material_interactions>** command tells the code how to handle inter-material and also material-surface interactions. Possible children include:
 - surface_hit: how to treat particles impacting surfaces. Support for fluid materials is pending.
 - mcc: MCC collisions for particle-fluid interactions
 - dsmc: DSMC collisions for particle-particle interactions
 - chemistry: fluid-fluid reactions that can be used to model ionization

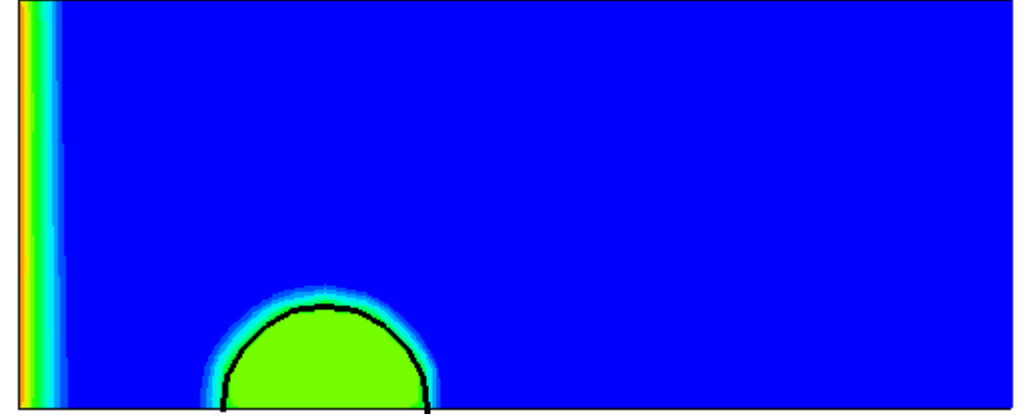
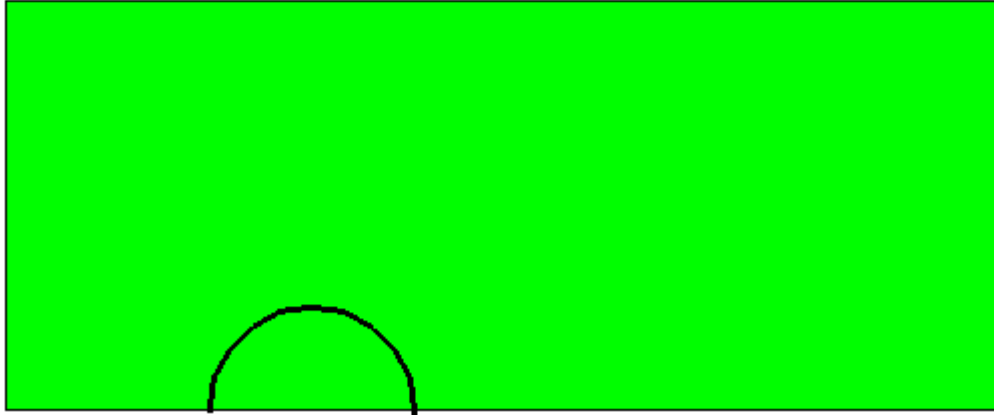
```
<material_interactions>
<chemistry process="ionization">
  <sources>Xe,e-</sources>
  <products>Xe+,e-</products>
  <rate>const</rate>
  <rate_coeffs>1e-18</rate_coeffs>
</chemistry>

<mcc process="scatter">
  <source>Xe+</source>
  <target>Xe</target>
  <sigma>const</sigma>
  <sigma_coeffs>1e-18</sigma_coeffs>
</mcc>

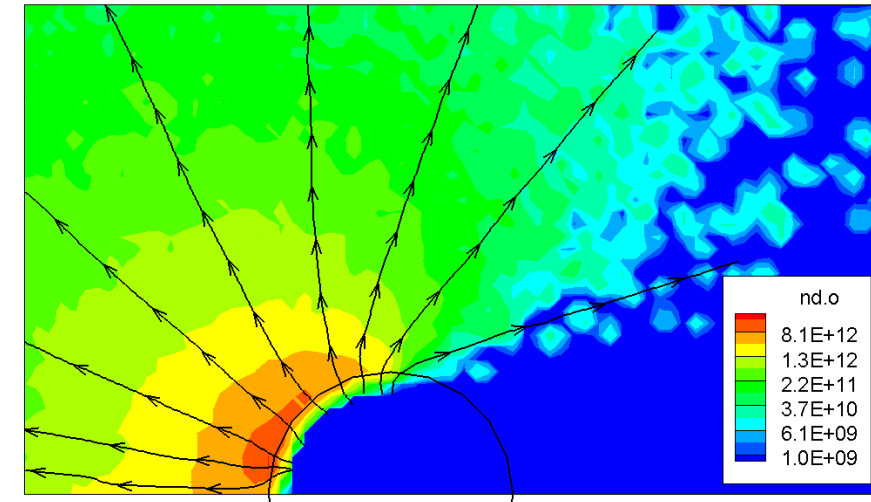
<surface_hit source="Xe+" target="BN">
  <emission>diffuse</emission>
  <product>Xe</product>
  <c_stick>0.5</c_stick>
  <c_rest>1</c_rest>
  <c_accom>0.5</c_accom>
  <sputter type="const" yield="0.1" product="BN" />
</surface_hit>
</material_interactions>
```

Results

- Here is an animation of ion number density and plasma potential for this case

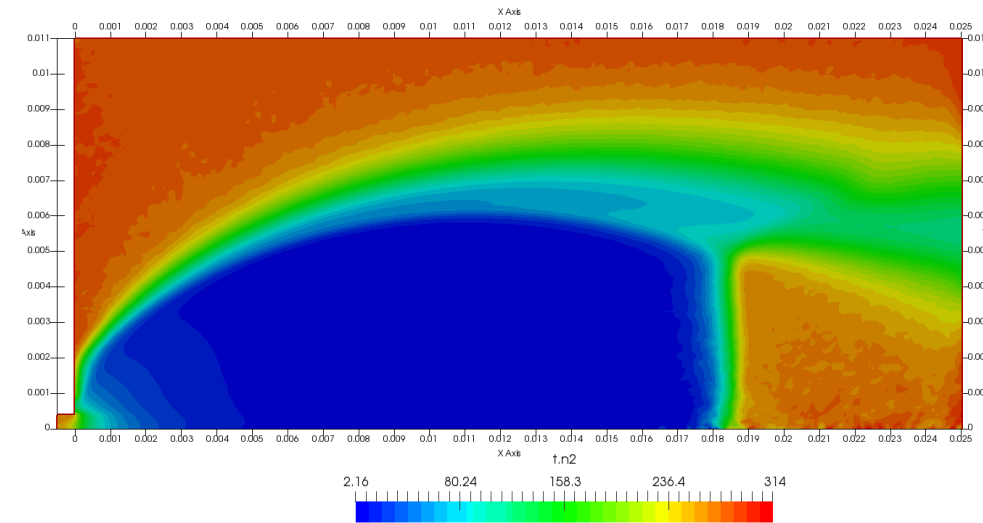
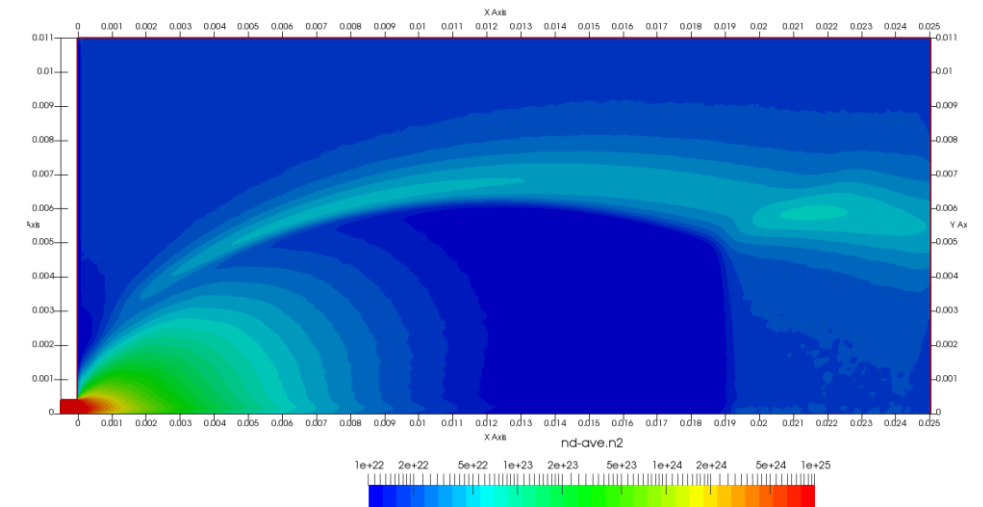
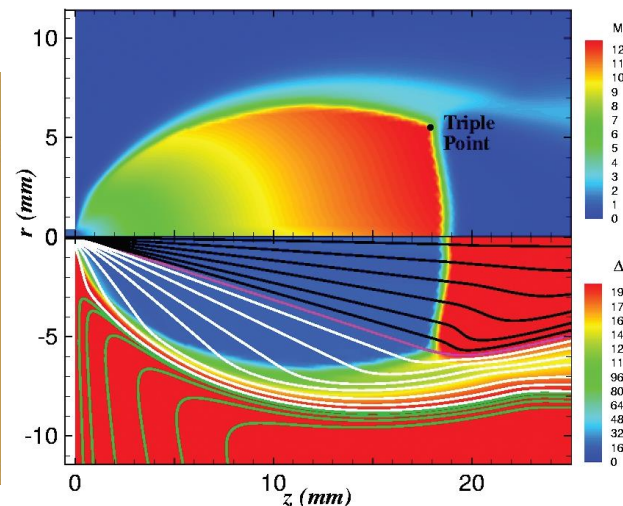
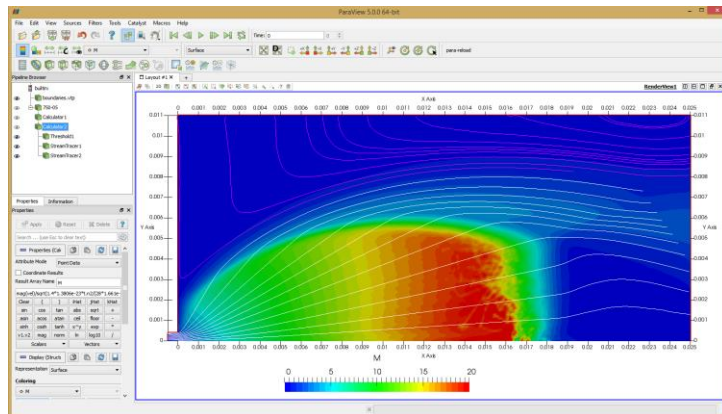


- We can also visualize number density of neutrals generated by ions recombining at the surface



DSMC example

- See <https://www.particleincell.com/2017/starfish-tutorial-dsmc/> for a DSMC example
- Simulates expansion of atmospheric pressure gas into a vacuum cavity
- Based on a CFD study by
 - Jugroot, M., Groth, C., Thomson B., Baranov V., Collings, B., Numerical investigation of interface region flows in mass spectrometers: neutral gas transport, J. of Phys., D: Applied Physics, vol. 37, pp. 1289–1300, 2004



DSMC

- To add DSMC, you need to add a **<dsmc>** material interaction
 - Next specify the two interacting materials in **<pair>**
 - VHS collision cross-section based on equation 4.63 in Bird is implemented
 - It uses material properties specified in **<materials>**

```

<!-- materials file -->
<materials>

  <material name="N2" type="kinetic">
    <molwt>28</molwt>
    <charge>0</charge>
    <spwt>1e11</spwt>
    <ref_temp>275</ref_temp>;
    <visc_temp_index>0.74</visc_temp_index>
    <vss_alpha>1.00</vss_alpha>
    <diam>4.17e-10</diam>
  </material>

  <material name="SS" type="solid">
    <molwt>52.3</molwt>
    <density>8000</density>
  </material>

</materials>
    
```

```

<!-- material interactions file -->
<material_interactions>
  <surface_hit source="N2" target="SS">
    <product>N2</product>
    <model>diffuse</model>
    <prob>1.0</prob>
  </surface_hit>

  <dsmc model="elastic">
    <pair>N2,N2</pair>
    <sigma>Bird463</sigma>
  </dsmc>

</material_interactions>
    
```

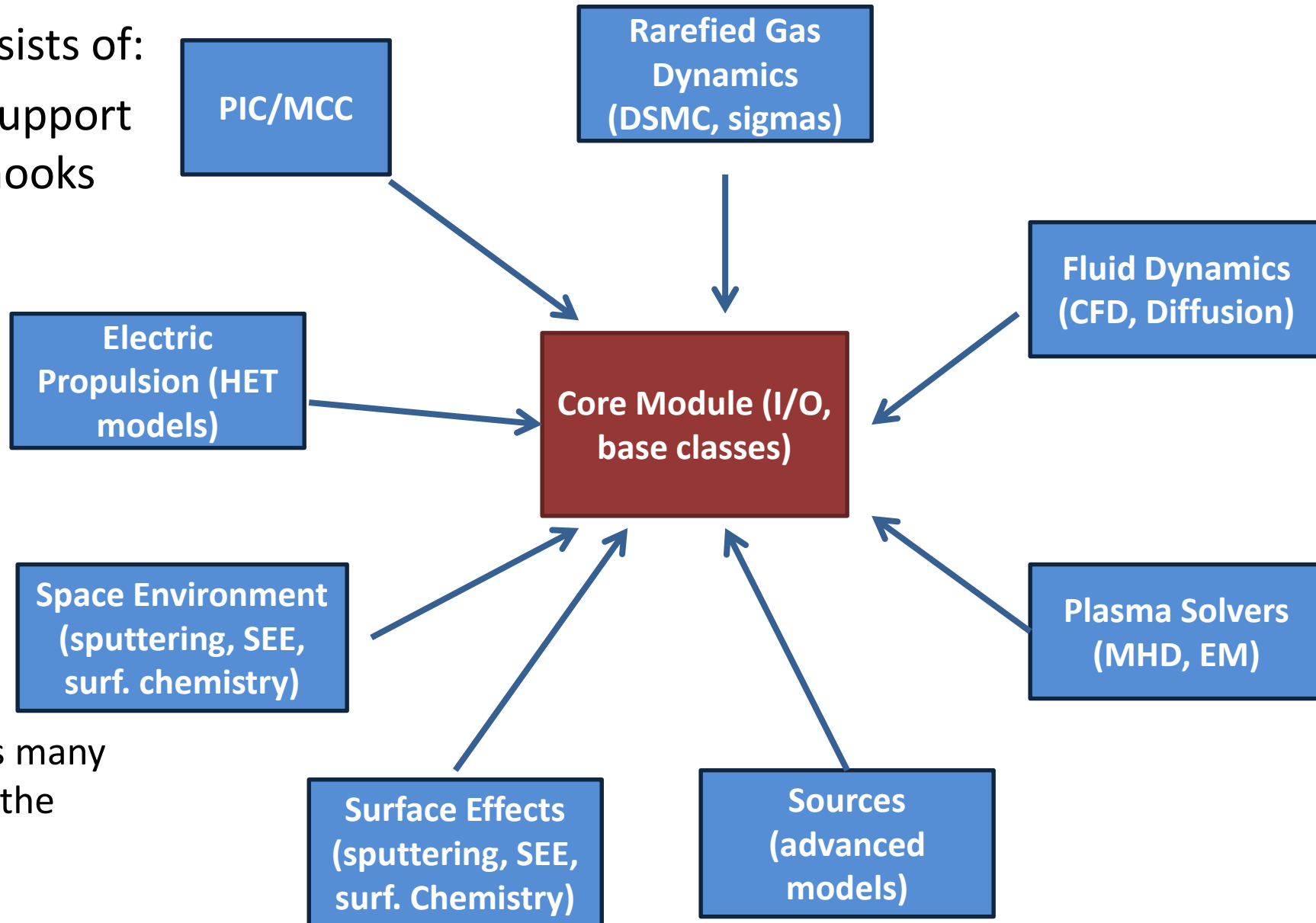
STARFISH DEVELOPMENT

Code Overview

- Starfish conceptually consists of:

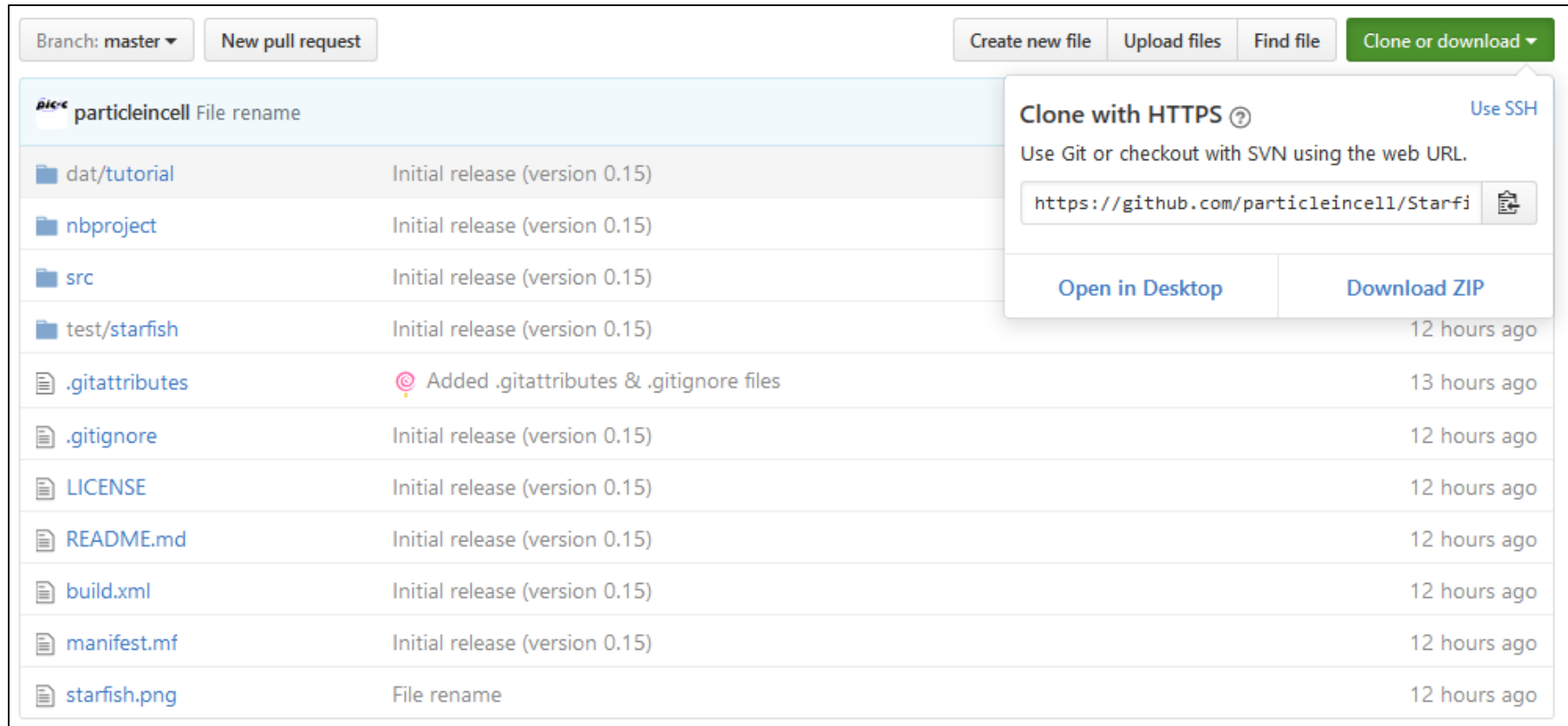
- Core** library providing support for I/O and main logic hooks
- Numerous **modules** implementing relevant physics

- The modules can be further extended with **plugins**
 - The full version implements many different plugins extending the Starfish-LE capabilities



Source Code

- Start by getting the source code from GitHub: <https://github.com/particleincell/Starfish-LE>



Branch: master ▾ New pull request

Create new file Upload files Find file Clone or download ▾

particleincell File rename

dat/tutorial	Initial release (version 0.15)	
nbproject	Initial release (version 0.15)	
src	Initial release (version 0.15)	
test/starfish	Initial release (version 0.15)	12 hours ago
.gitattributes	Added .gitattributes & .gitignore files	13 hours ago
.gitignore	Initial release (version 0.15)	12 hours ago
LICENSE	Initial release (version 0.15)	12 hours ago
README.md	Initial release (version 0.15)	12 hours ago
build.xml	Initial release (version 0.15)	12 hours ago
manifest.mf	Initial release (version 0.15)	12 hours ago
starfish.png	File rename	12 hours ago

Clone with HTTPS ? Use SSH

Use Git or checkout with SVN using the web URL.

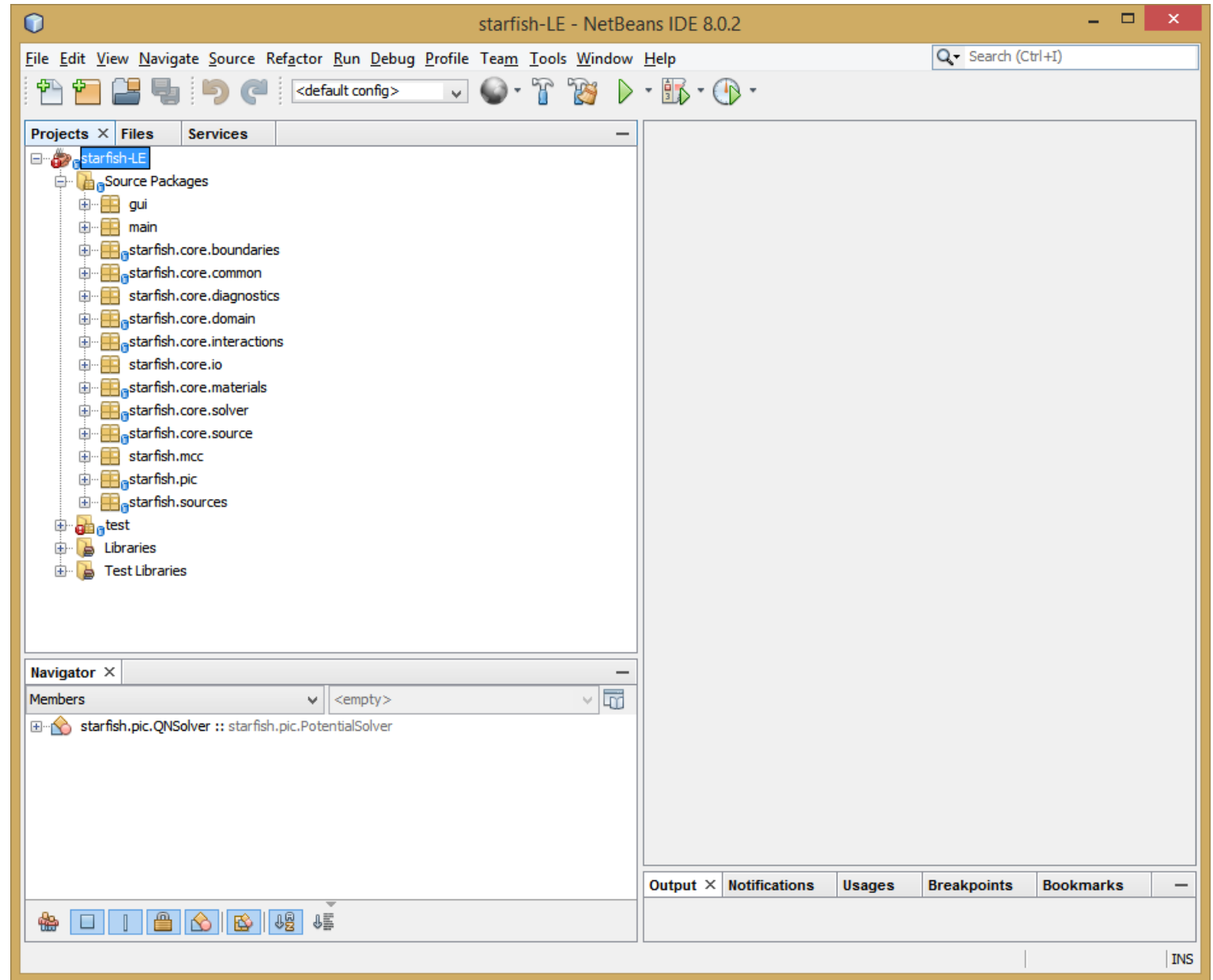
<https://github.com/particleincell/Starfish-LE>

Open in Desktop Download ZIP

- While you can just download a zip file of the entire source, it's better to clone the repo using Git. This will make it easier to receive updates and makes it possible for you to contribute to the project
 - <http://stackoverflow.com/questions/5989893/github-how-to-checkout-my-own-repository#5989998>
 - GitHub Desktop provides easy to use graphical interface to Git: <https://desktop.github.com/>

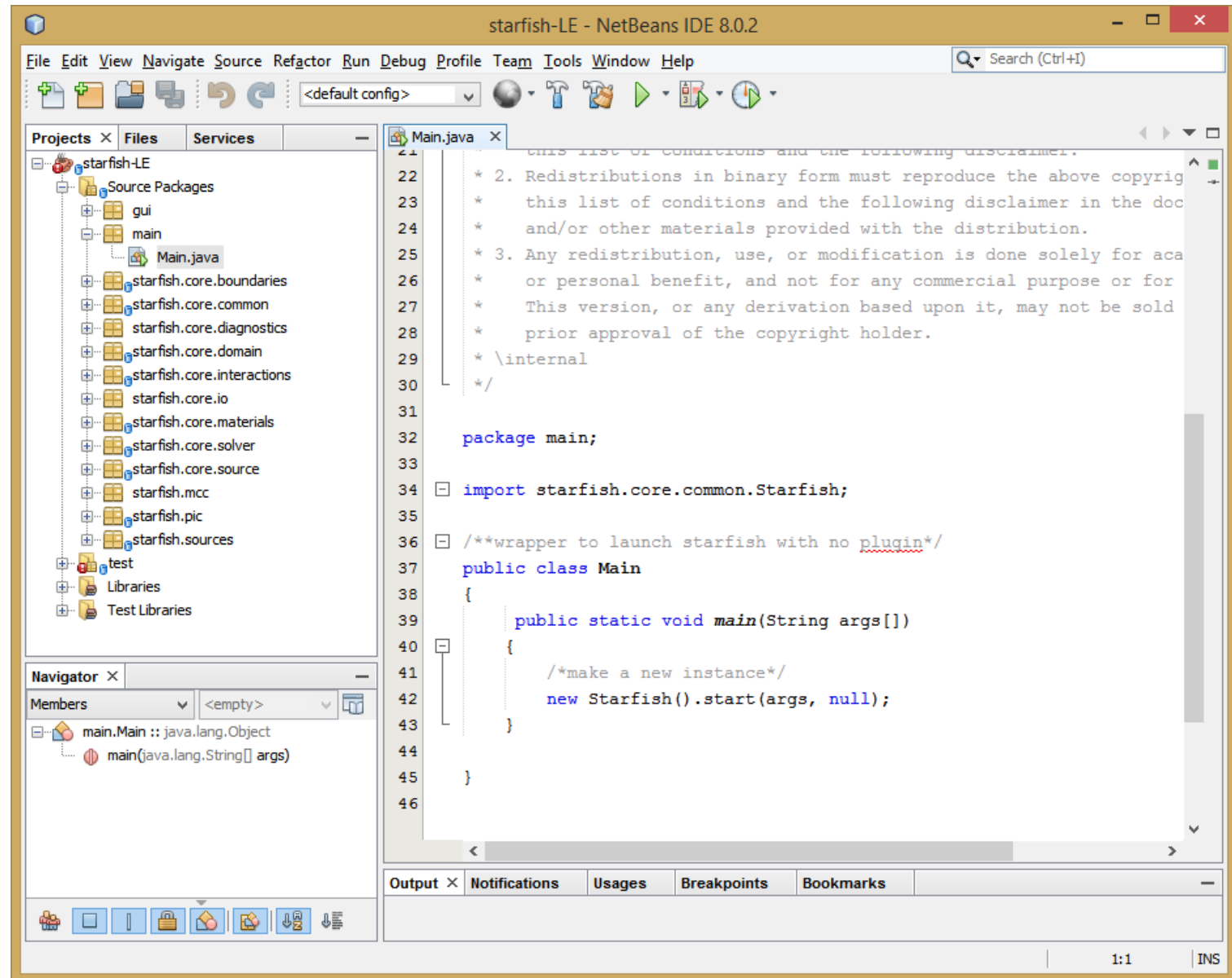
Netbeans

- I use Netbeans for the development environment but Eclipse should work just as well
- This image shows the package layout



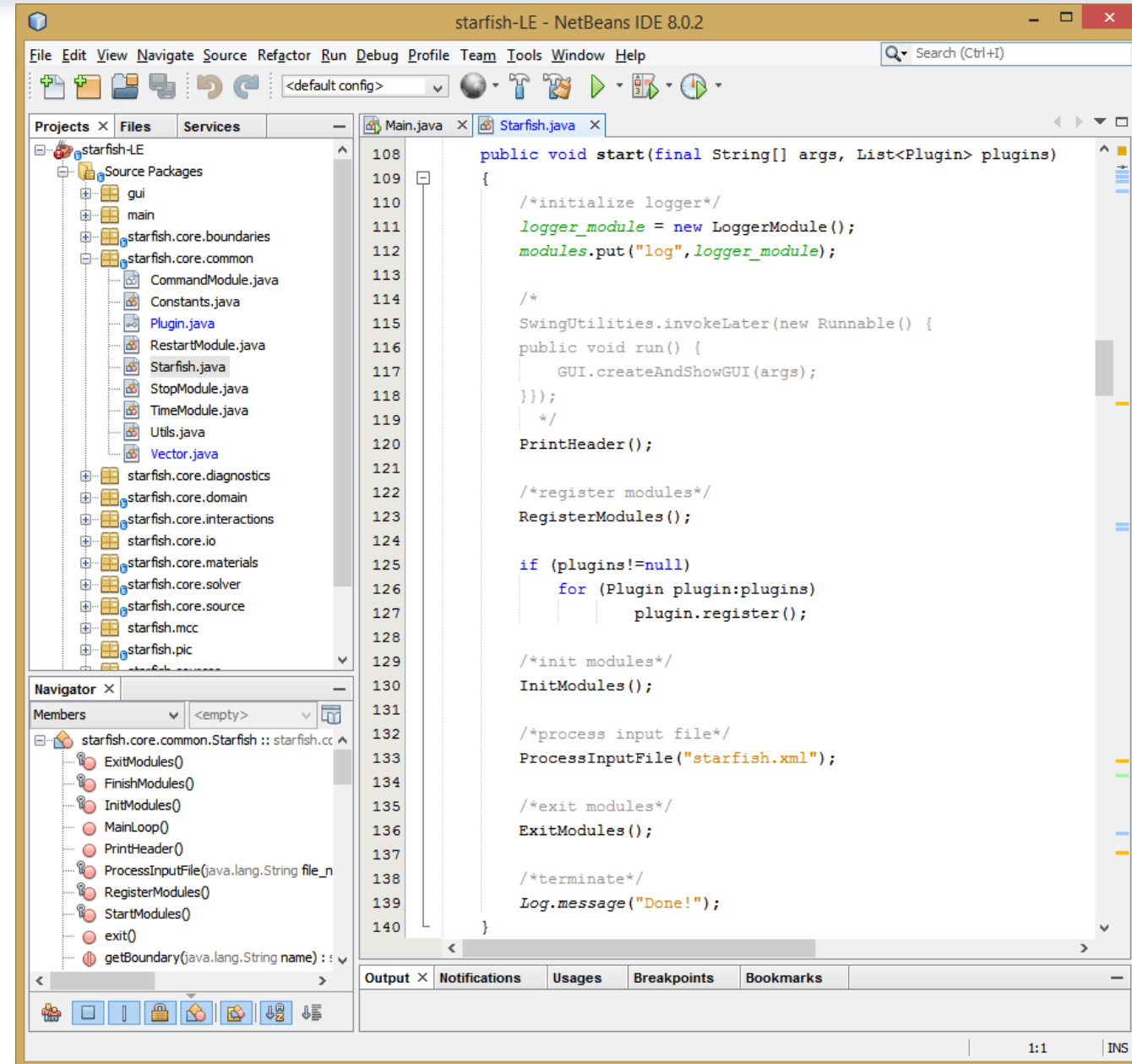
Main

- Just as any Java program, Starfish execution begins in a function called “main”
- This function is defined in Main.java located in package **main**
- This function instantiates a new object of type Starfish and then calls that object’s **start** method.



Start

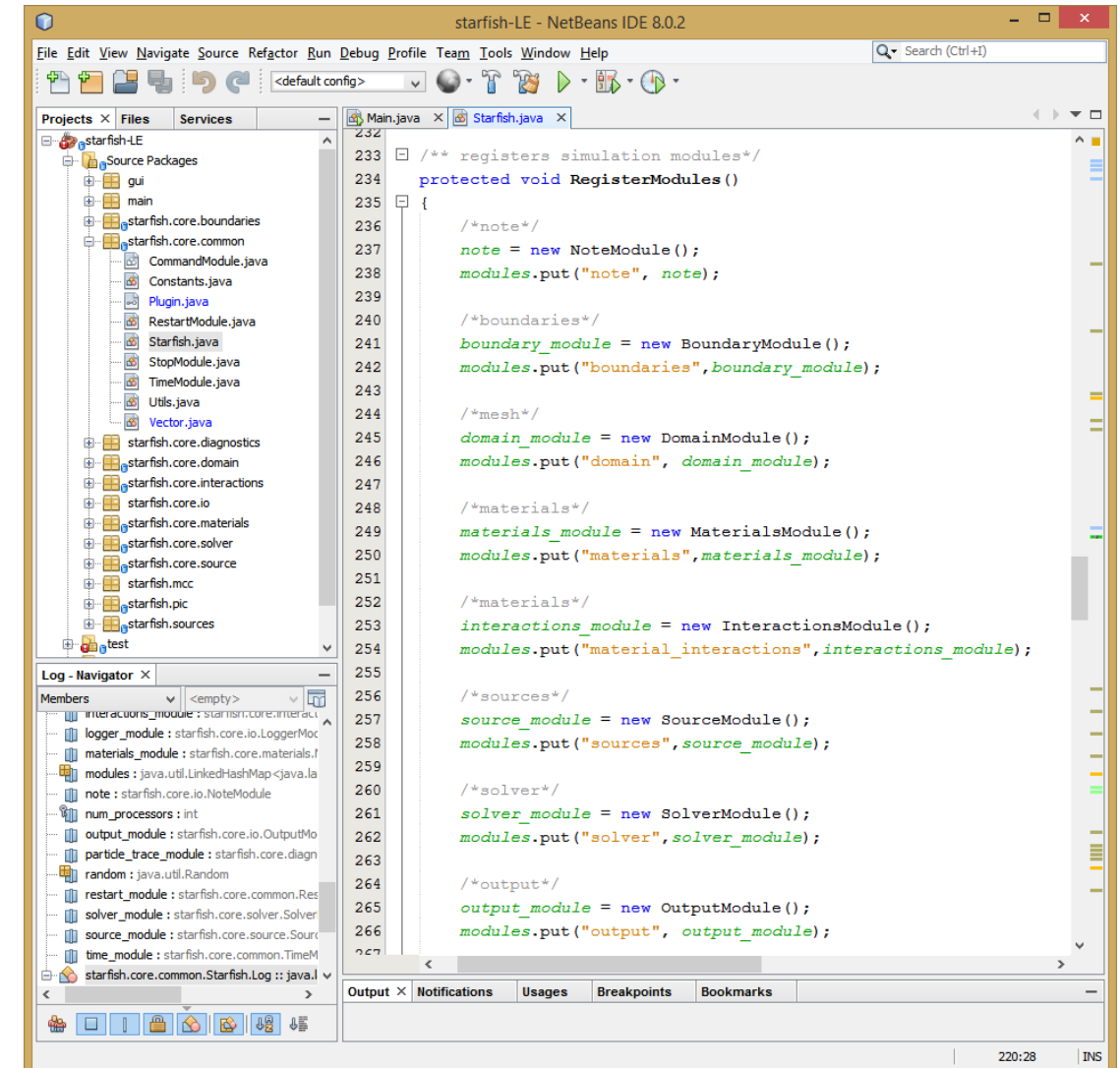
- This **start** method is located in Starfish.java located in package **starfish.core.common**
- It starts by instantiating a **LoggerModule**. This logger is how the rest of code prints information and error messages
- Next the header with version and copyright info is printed
- Next all default modules are registered
- Plugins are registered next, if any
- The modules are then initialized
- The code then reads file called **starfish.xml** and performs commands as specified – this is the “meat” of the simulation
- ExitModules let’s modules perform clean up actions
- Finally, “Done” is printed to the screen



RegisterModules

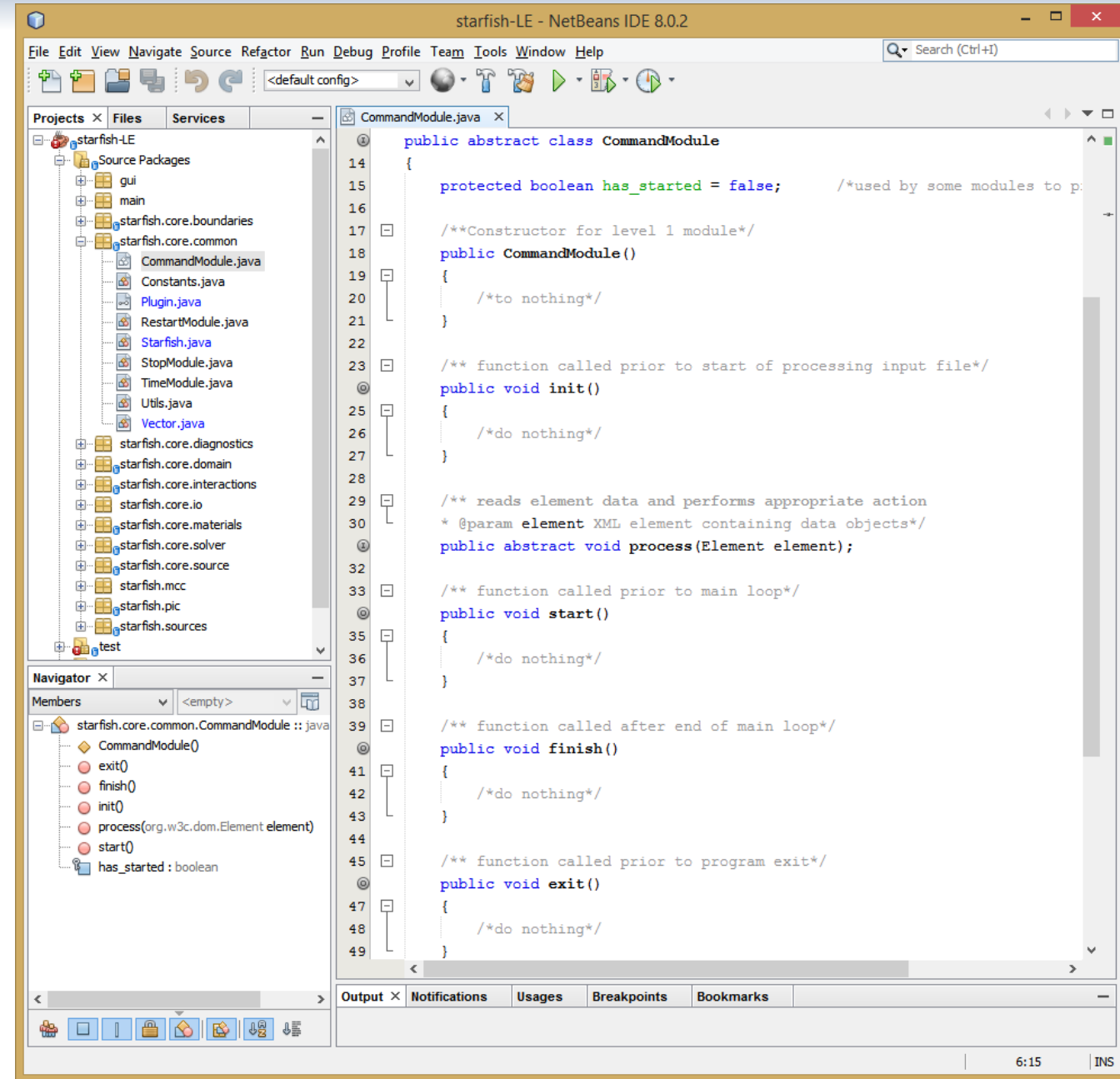
```
/*iterable list of registered modules, using LinkedHashMap to get predictable ordering*/  
static LinkedHashMap<String,CommandModule> modules = new LinkedHashMap<String,CommandModule>();
```

- Starfish modules are stored in a HashMap called modules. The accessor (key) is a string identifying the module name. All modules are derived from base class **CommandModule**
- **RegisterModules** is also defined in Starfish.java
- As you can see, this function simply adds (using put) various modules to the hash map
 - Some modules are first instantiated into a member variable – this is so other functions can use these modules without going through the hash map



Intro to Modules

- All modules extend from base class **CommandModule**
 - In starfish.core.common
- This base class defines five functions that need to be overloaded as needed:
 - **process**: called when command tag is encountered in starfish.xml input file
 - **init**: called by InitModules before simulation main loop starts
 - **start**: called at the start of the main loop
 - **finish**: called at the end of the main loop
 - **exit**: called by ExitModules just prior to code termination
- Why two initialization functions?
 - Some modules depend on others – for instance material interactions module needs material list to be initialized



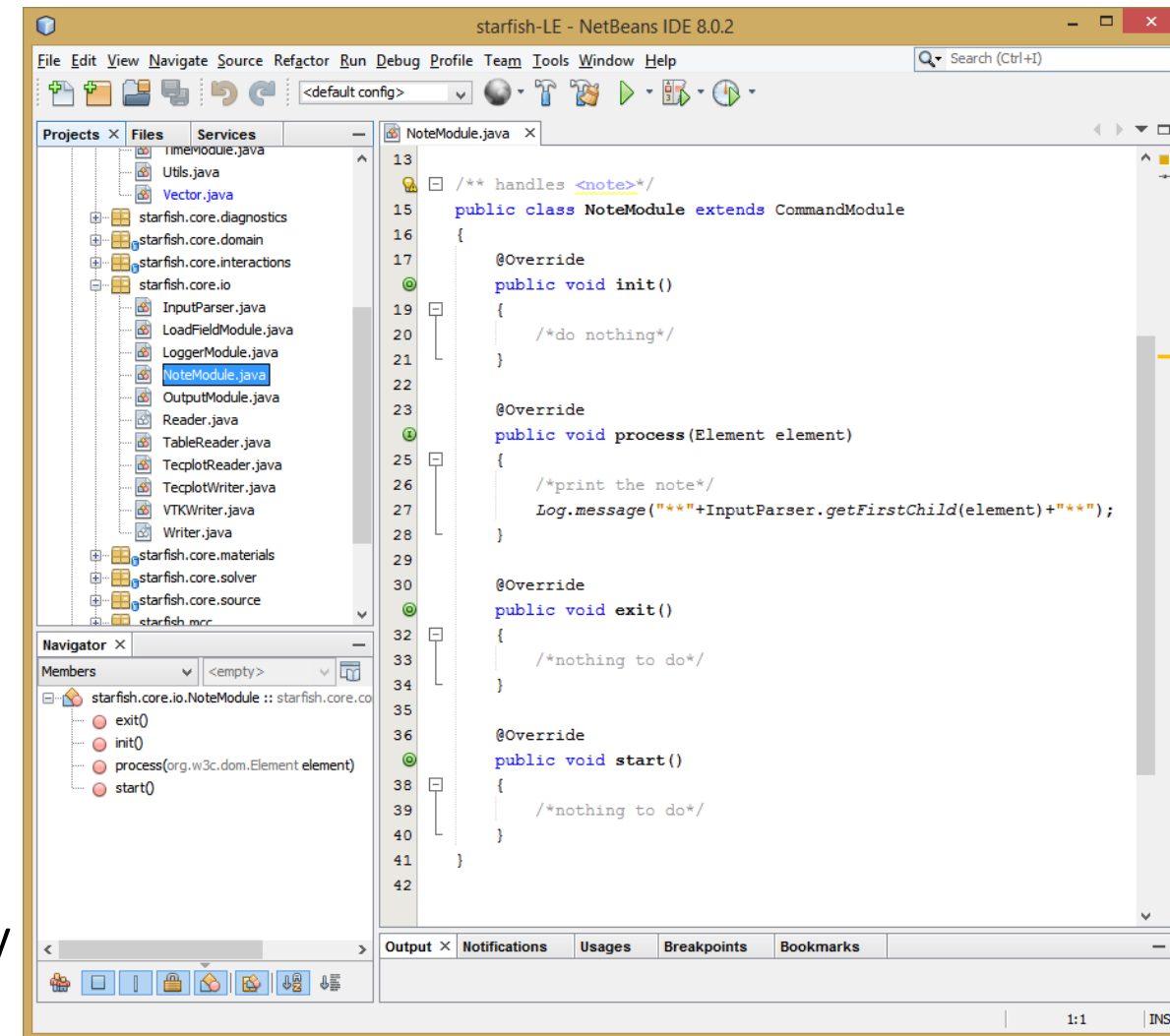
Example: Note Module

- As an example, let's take a look at the Note Module
- This module's process function is called when **<note>** tag is encountered in starfish.xml

```
<simulation>
<note>Starfish Tutorial: Part 1</note>
```

- Only the process method does anything, and that's simply to call **Log.message(..)** with the message given by **InputParser.getFirstChild(element)**
 - In this case this will be "Starfish Tutorial: Part 1"
- The argument to the process method of all modules is the XML element for the handled tag
 - This element can contain many additional child tags as well as attributes. **InputParser** class provides handy accessor methods

```
/*note*/
note = new NoteModule();
modules.put("note", note);
```

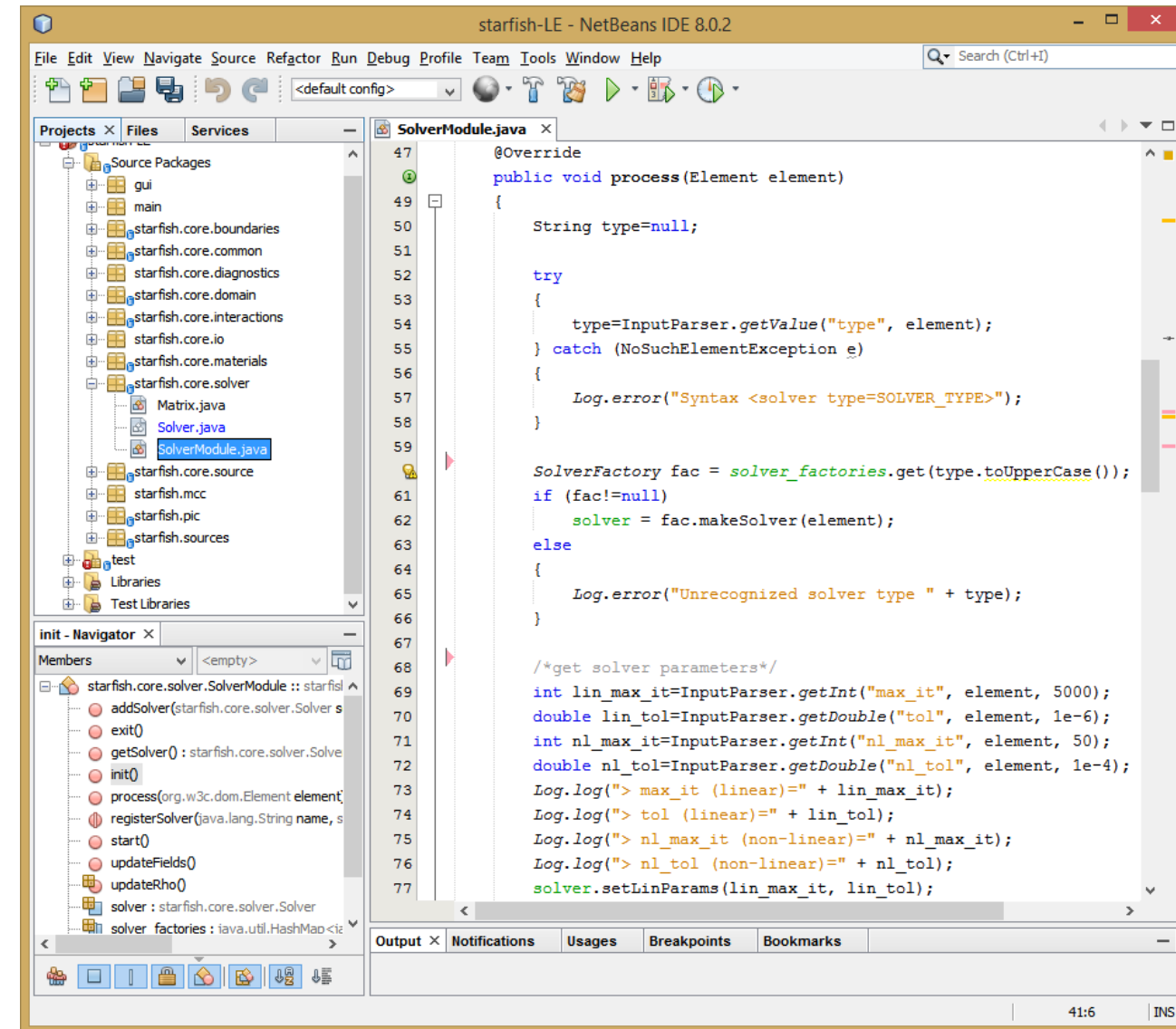


Second Example: Solver

- We now consider a more complex example: the `<solver>` tag
 - In `SolverModule.java` in `starfish.core.solver`

```
<!-- set potential solver -->
<solver type="poisson">
  <n0>1e12</n0>
  <Te0>1.5</Te0>
  <phi0>0</phi0>
  <max_it>1e4</max_it>
  <nl_max_it>25</nl_max_it>
  <tol>1e-4</tol>
  <nl_tol>1e-3</nl_tol>
  <linear>false</linear>
</solver>
```

- `InputParser.getValue/getInt/getDouble...` provide easy way to obtain data regardless of whether it was defined as an *attribute* (type) or *child elements* (n0, Te0...)



Solver Module

- Starfish implements modularity using object oriented programming
- You already saw an example with the modules extending base CommandModule
- Another example is Solver module
- The string provided for “type” is used to retrieve a **SolverFactory** from a hash map
- This factory then generates a solver object that derives from the base **Solver** class
- At no point does the actual solver module know (or care!) what type of a solver the user specified
- Different solver types are registered in **init**
 - You may want to write a **plugin** to register additional solvers

```
try
{
    type=InputParser.getValue("type", element);
} catch (NoSuchElementException e)
{
    Log.error("Syntax <solver type=SOLVER_TYPE>");
}

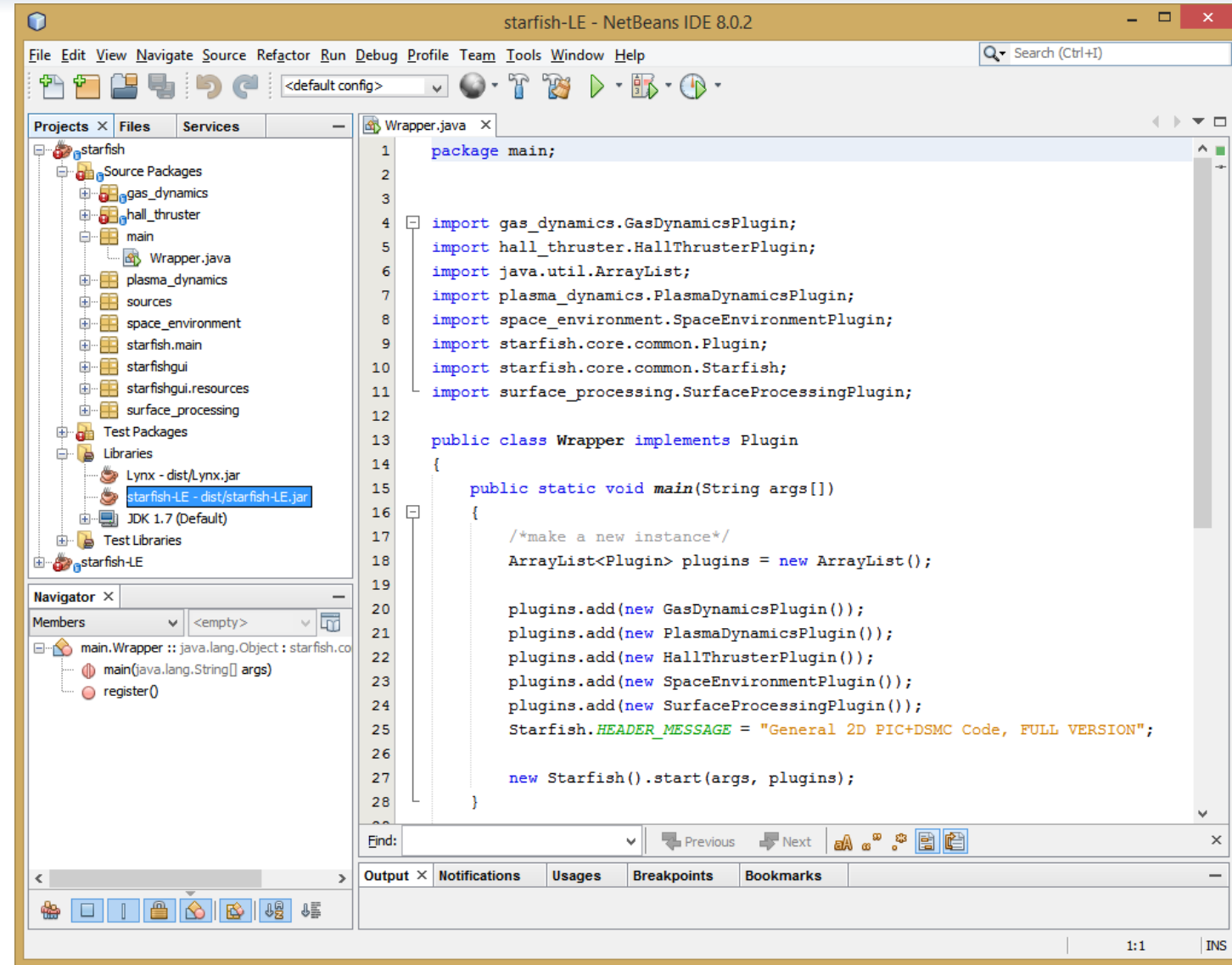
SolverFactory fac = solver_factories.get(type.toUpperCase());
if (fac!=null)
    solver = fac.makeSolver(element);
else
{
    Log.error("Unrecognized solver type " + type);
}
```

```
@Override
public void init()
{
    registerSolver("CONSTANT-EF", ConstantEF.constantEFSolverFactory);
    registerSolver("QN", QNSolver.boltzmannSolverFactory);
    registerSolver("POISSON", PoissonSolver.poissonSolverFactory);
}
```

Plugins

- The easiest way to extend Starfish LE is using **plugins**
 - This is in fact how the full version Starfish code works
- Create a new project and add starfish-LE as a dependency under Libraries
- Define new “main” in which you assemble an ArrayList of **Plugin**, then pass this list to **Starfish().start**
- **Register** method of each plugin will be called after **RegisterModules**
 - Your plugin could for instance call

```
SolverModule.registerSolver("my_solver", mySolverFactory);
```



Plugins

- Here is actual example of the gas dynamics plugin from the full version
 - It registers new material type, based on the diffusion equation
 - Registers DSMC as new material interaction
 - Also registers new collision cross-section sigma

```
package gas_dynamics;

import starfish.core.common.Plugin;
import starfish.core.interactions.InteractionsModule;
import starfish.core.materials.MaterialsModule;

public class GasDynamicsPlugin implements Plugin
{
    @Override
    public void register()
    {
        /*add new material types*/
        MaterialsModule.registerMaterialType("FLUID_DIFFUSION", FluidDiffusion.FluidDiffusionParser);

        /*add new interactions*/
        InteractionsModule.registerInteraction("DSMC", DSMC.DSMCFactory);

        /*add cross-section*/
        InteractionsModule.registerSigma("Bird463", SigmaPlus.makeSigmaBird463);
    }
}
```

Solvers

- The SolverModule contains function called **updateFields**
- This function is called by the Starfish main loop at every time step. The function in turn calls **update** for the defined solver type
- We will now take a look at the simplest solver used in PIC, quasi neutral Boltzmann inversion, $\phi = \phi_0 + kT_{e,0} \ln\left(\frac{n_i}{n_0}\right)$
- Defined in QNSolver.java in starfish.pic
- The factory reads in appropriate values from the input file and instantiates object of type QNSolver (derived from Solver)
- It then returns this object

```
public void updateFields()
{
    /*update rho*/
    updateRho();

    /*call solver*/
    solver.update();

    /*update electric field*/
    solver.updateGradientField();
}
```

```
public static SolverModule.SolverFactory boltzmannSolverFactory = new SolverModule.SolverFactory()
{
    @Override
    public Solver makeSolver(Element element)
    {
        double n0=InputParser.getDouble("n0", element);
        double Te0=InputParser.getDouble("Te0", element);
        double phi0=InputParser.getDouble("phi0", element);
        Solver solver=new QNSolver(n0, phi0, Te0);

        /*log*/
        Starfish.Log.log("Added BOLTZMANN solver");
        Starfish.Log.log("> n0  =" + n0 + " (#/m^3)");
        Starfish.Log.log("> T0  =" + Te0 + " (eV)");
        Starfish.Log.log("> phi0 =" + phi0 + " (V)");

        return solver;
    }
};
```

QN Solver

- Starfish stores mesh-based quantities in object of type Field2D. This object defines functions for interpolation and also getData which returns double[][] containing the actual node values.
- The domain module automatically generates fields to store charge density ρ and potential ϕ
- Each mesh node is also classified as DIRICHLET, OPEN, etc...

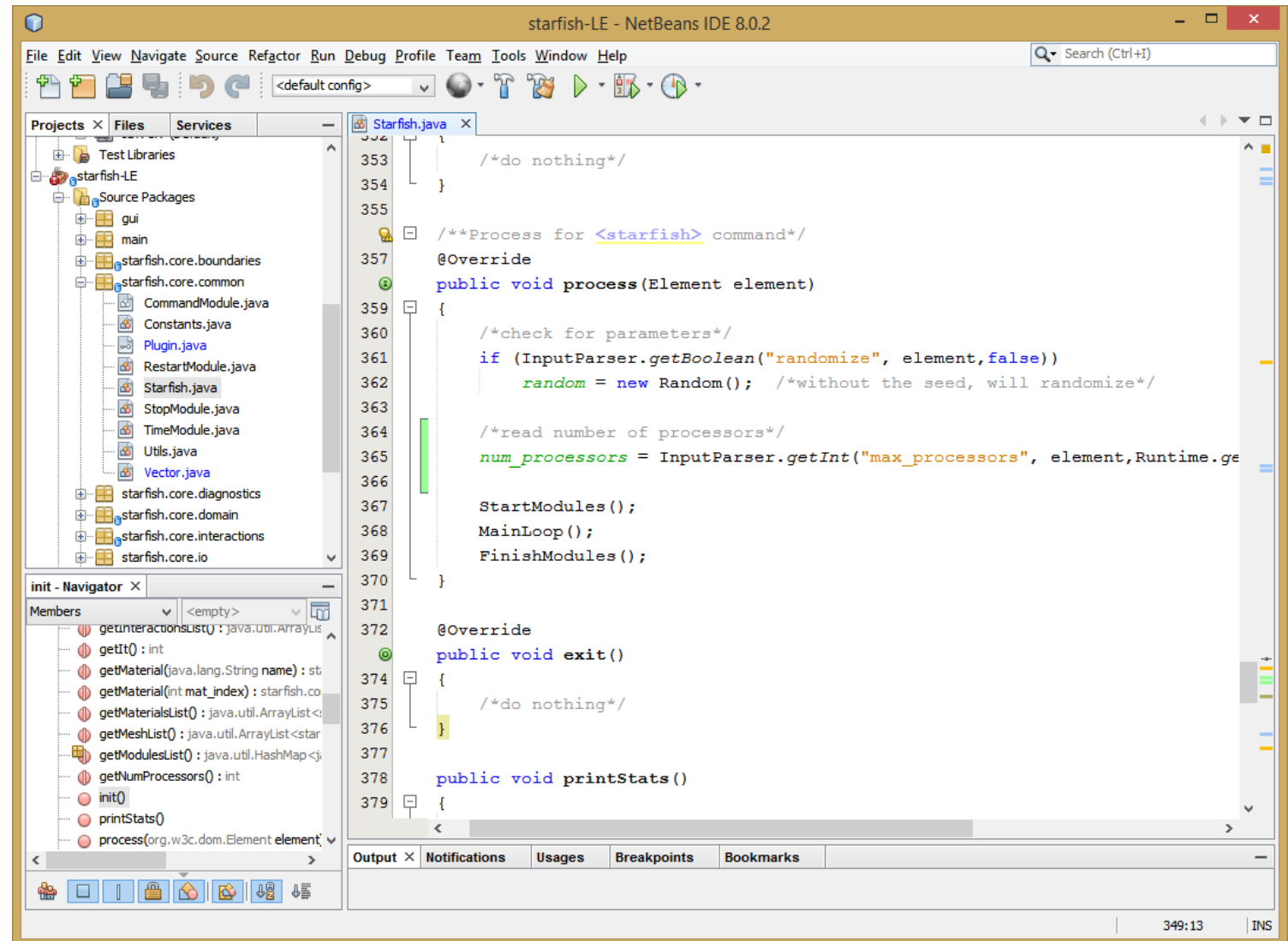
```
@Override
public void update()
{
    for (Mesh mesh:Starfish.getMeshList())
    {
        int ni = mesh.ni;
        int nj = mesh.nj;
        double phi[][] = Starfish.domain_module.getPhi(mesh).getData();
        double rho[][] = Starfish.domain_module.getRho(mesh).getData();

        for (int i=0;i<ni;i++)
            for (int j=0;j<nj;j++)
            {
                if (mesh.nodeType(i, j) == NodeType.DIRICHLET)
                    continue;

                double ion_den = rho[i][j]/Constants.QE;
                if (ion_den>0)
                    phi[i][j] = phi0 + kTe0*Math.log(ion_den/den0);
                else
                    phi[i][j] = phi0 + kTe0*Math.log(1e-10);          /*bad
            }
        }
    }
}
```

Starfish Module

- The simulation is started by the **process** function of StarfishModule
 - Handles <starfish> tag
- First calls **start** on all modules
- The simulation main loop then starts
- The **finish** function is then called



Main Loop

- The main loop performs the typical operations expected in a PIC/DSMC code:
 - Mass is injected into the simulation domain
 - Densities of different material species are computed at a new time step
 - Interactions between different materials are considered
 - Fields are updated to compute forces
 - Restart data is saved as needed
 - Averaging and file output is also performed as needed
- The above steps repeated until some stopping condition
 - Maximum number of time steps is reached
 - Simulation reaches steady state

Main Loop

- This is what it looks like in practice

```
/**simulation main loop*/
public void MainLoop()
{
    Log.message("Starting main loop");

    restart_module.load();

    /*compute initial field*/
    solver_module.updateFields();

    while(time_module.hasTime())
    {
        /*add new particles*/
        source_module.sampleSources();

        /*update densities and velocities*/
        materials_module.updateMaterials();

        /*perform material interactions (collisions and
        interactions_module.performInteractions();

        /*solve potential and recompute electric field*/
        solver_module.updateFields();
```

```
        /*save restart data*/
        restart_module.save();

        /*save animations*/
        animation_module.save();

        /*save average data*/
        averaging_module.sample();

        printStats();

        /*advance time*/
        time_module.advance();
    }/*end of main loop*/

    /*save average data*/
    averaging_module.sample();

    /*check if we have reached the steady state*/
    if (!time_module.steady_state)
        Log.warning("The simulation failed to reach
        steady state!");
```

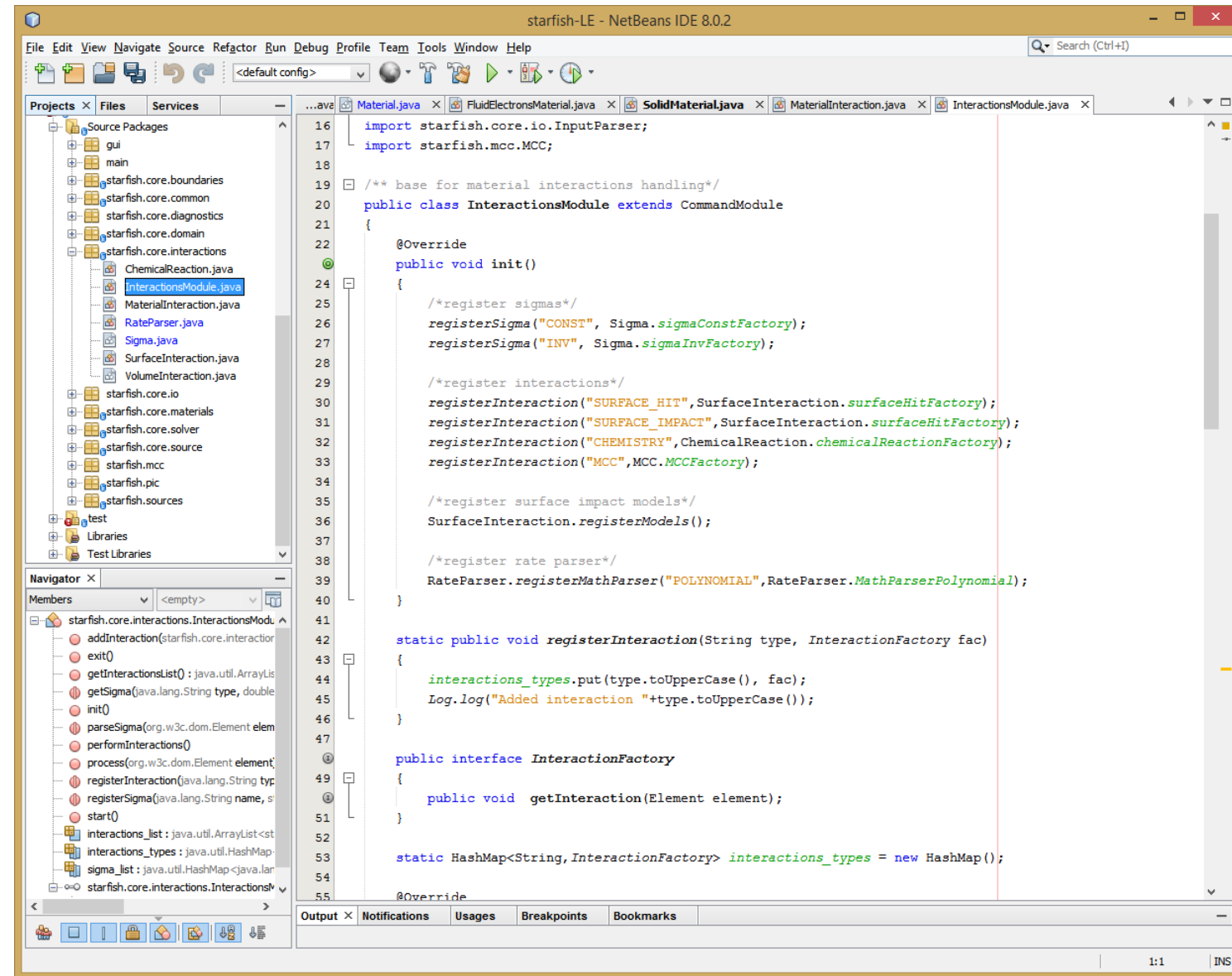
Materials

- The **updateMaterials** function is another example of modularity afforded by object oriented programming
- While Starfish is mainly used for kinetic simulations, the code is not hardwired for this
- Even in a PIC simulation, we merely care about the **density** (and possibly velocities/temperature) of different species, $\rho = \sum_i q_i n_i$
 - The kinetic push of particles $\vec{x}^{k+1} = \vec{x}^k + \vec{v}^{k+0.5} \Delta t$ is only an intermediary step to compute n^{k+1}
- Starfish implements this abstraction in the sense that every material type implements an **update** function which computes the new density, temperature, and bulk velocities at the new time step
 - For kinetic materials, this implies performing the push followed by scatter
 - For fluid materials, this may imply advancing Navier-Stokes solutions forward by Δt

```
/**updates densities of all flying materials*/  
public void updateMaterials()  
{  
    for (Material mat:materials_list)  
        mat.update();  
}
```

Material Interactions

- Interaction between different species is handled by InteractionModule
- Starfish-LE supports three types of interactions: surface impact, chemistry, and MCC
- Full version adds DSMC
- **Surface impact** is an interaction between a material (kinetic or fluid) and a surface boundary: example would be surface recombination or sputtering
- **Chemistry** is a fluid-fluid type interaction. Only the density fields (may be computed from particles) come to play and are used to compute rate constants. Example would be ionization.
- **MCC** is a particle-fluid interaction. The source material must be kinetic and the target is not affected.



The screenshot displays the NetBeans IDE 8.0.2 interface for the 'starfish-LE' project. The left sidebar shows the 'Projects' and 'Files' views. The 'Files' view highlights the 'starfish.core.interactions' package, with 'InteractionsModule.java' selected. The 'Members' view below it lists the methods and fields of the 'InteractionsModule' class. The main editor window shows the code for 'InteractionsModule.java', which extends 'CommandModule'. The code includes imports for 'InputParser' and 'MCC', a comment indicating it's a base for material interactions, and an '@Override' method 'init()' that registers various interactions and sigma factories. It also includes a static method 'registerInteraction' and an 'InteractionFactory' interface.

```
16 import starfish.core.io.InputParser;
17 import starfish.mcc.MCC;
18
19 /** base for material interactions handling*/
20 public class InteractionsModule extends CommandModule
21 {
22     @Override
23     public void init()
24     {
25         /*register sigmas*/
26         registerSigma("CONST", Sigma.sigmaConstFactory);
27         registerSigma("INV", Sigma.sigmaInvFactory);
28
29         /*register interactions*/
30         registerInteraction("SURFACE_HIT", SurfaceInteraction.surfaceHitFactory);
31         registerInteraction("SURFACE_IMPACT", SurfaceInteraction.surfaceHitFactory);
32         registerInteraction("CHEMISTRY", ChemicalReaction.chemicalReactionFactory);
33         registerInteraction("MCC", MCC.MCCFactory);
34
35         /*register surface impact models*/
36         SurfaceInteraction.registerModels();
37
38         /*register rate parser*/
39         RateParser.registerMathParser("POLYNOMIAL", RateParser.MathParserPolynomial);
40     }
41
42     static public void registerInteraction(String type, InteractionFactory fac)
43     {
44         interactions_types.put(type.toUpperCase(), fac);
45         Log.log("Added interaction "+type.toUpperCase());
46     }
47
48     public interface InteractionFactory
49     {
50         public void getInteraction(Element element);
51     }
52
53     static HashMap<String, InteractionFactory> interactions_types = new HashMap();
54
55     @Override
```


Surface Interactions

- The division on the previous page is actually bit of a simplification. Material Interactions are actually grouped into **surface interactions** and **volume interactions**. MCC, DSMC, and Chemistry are subclasses of VolumeInteraction since they deal with effects within the computational mesh (volume).
- Surface interactions are currently implemented only for kinetic materials and are processed when particle hits a surface
 - In KineticMaterial.java and Material.java

```
boolean ProcessBoundary(Particle part, Mesh mesh, double old[],
{
    boolean left_mesh = false;
    Face exit_face = null;
```

```
/*perform surface interaction*/
boolean alive = false;
if (target_mat!=null)
    alive = target_mat.performSurfaceInteraction(part.vel, mat_index, seg_min, tsurf_min);
```

```
boolean performSurfaceInteraction(double[] vel, int source_index, Segment segment, double t)
{
    /*first check for sputtering or surface emission hooks*/
    ArrayList<MaterialInteraction> emission = target_interactions.getInteractionList(source_index);
    for (MaterialInteraction em:emission)
    {
        em.callSurfaceImpactHandler(vel, segment, t);
    }
}
```

Volume Interactions

- Volume interactions instead handled by MaterialInteractions.performInteraction. This function called from the main loop.
- This functions iterates over an ArrayList of volume interactions. New interactions can be added using **addInteraction** (for instance from a plugin)

```
public void addInteraction(VolumeInteraction handler)
{
    interactions_list.add(handler);
}
```

- As an example of a volume interaction, consider MCC shown on right
- Perform iterates over all particles and computes collision probability from

$$P = 1 - \exp(-\sigma v_{rel} n_a \Delta t)$$

```
/*performs material interactions*/
public void performInteractions()
{
    for (VolumeInteraction vint:interactions_list)
        vint.perform();
}
```

```
/*performs collisions on a single mesh*/
void perform(Mesh mesh)
{
    Iterator<Particle> iterator = source.getIterator(mesh);
    Field2D target_den = target.getDen(mesh);

    Field2D real_sum = fc_real_sum.getField(mesh);
    Field2D count_sum = fc_count_sum.getField(mesh);
    double dt = frequency*Starfish.getDt();

    //loop over particles
    while (iterator.hasNext())
    {
        Particle part = iterator.next();

        double den_a = target_den.gather(part.lc);
        if (den_a<=0) continue;

        /*create random target particle according to target T and stream velocity*/
        double target_vel[] = target.sampleMaxwellianVelocity(mesh,part.lc, 50);
        //double target_vel[] = target.sampleVelocity(mesh,part.lc);

        double g_vec[] = new double[3];
        for (int i=0;i<3;i++) g_vec[i] = target_vel[i] - part.vel[i];
        double g = Vector.mag3(g_vec);

        /*collision probability*/
        double P = 1-Math.exp(-sigma.eval(g)*g*dt*den_a);
```

Conclusion

- Please visit <https://www.particleincell.com/starfish/> for more information
- Don't hesitate to contact me at lubos.brieda@particleincell.com if you have questions
- If anything here is not clear, I suggest you take one of my past or upcoming plasma simulation courses: <https://www.particleincell.com/courses/>