Starfish Webinar
Source Code Overview

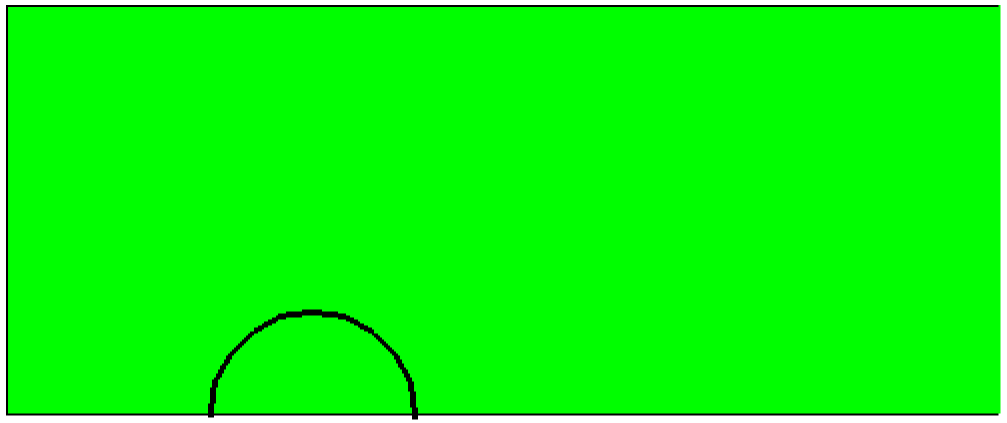
Lubos Brieda, Ph.D.
Particle In Cell Consulting LLC

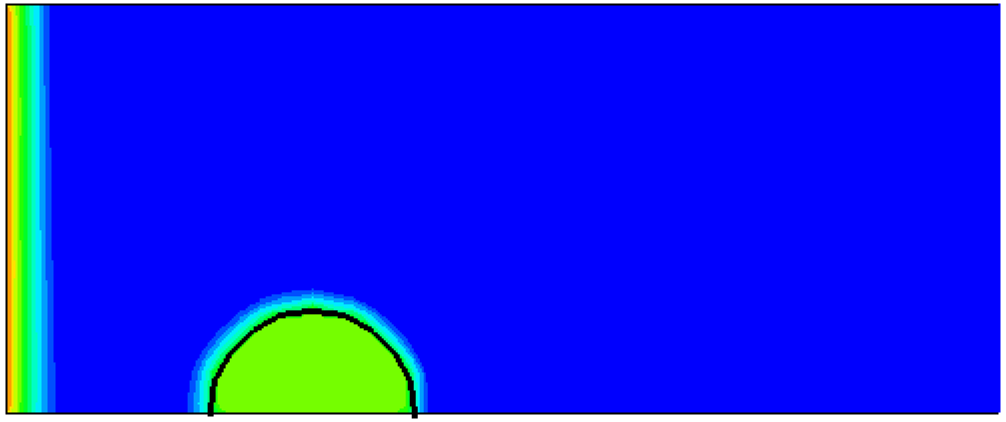
particleincell.com/starfish

Introduction

- What is Starfish?
  - Two-Dimensional (XY or RZ) Java code for modeling ionized and non-ionized gases
    - Currently mainly a rarefied plasma / gas code but plan is to add support for fluid modeling
  - 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
  - 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

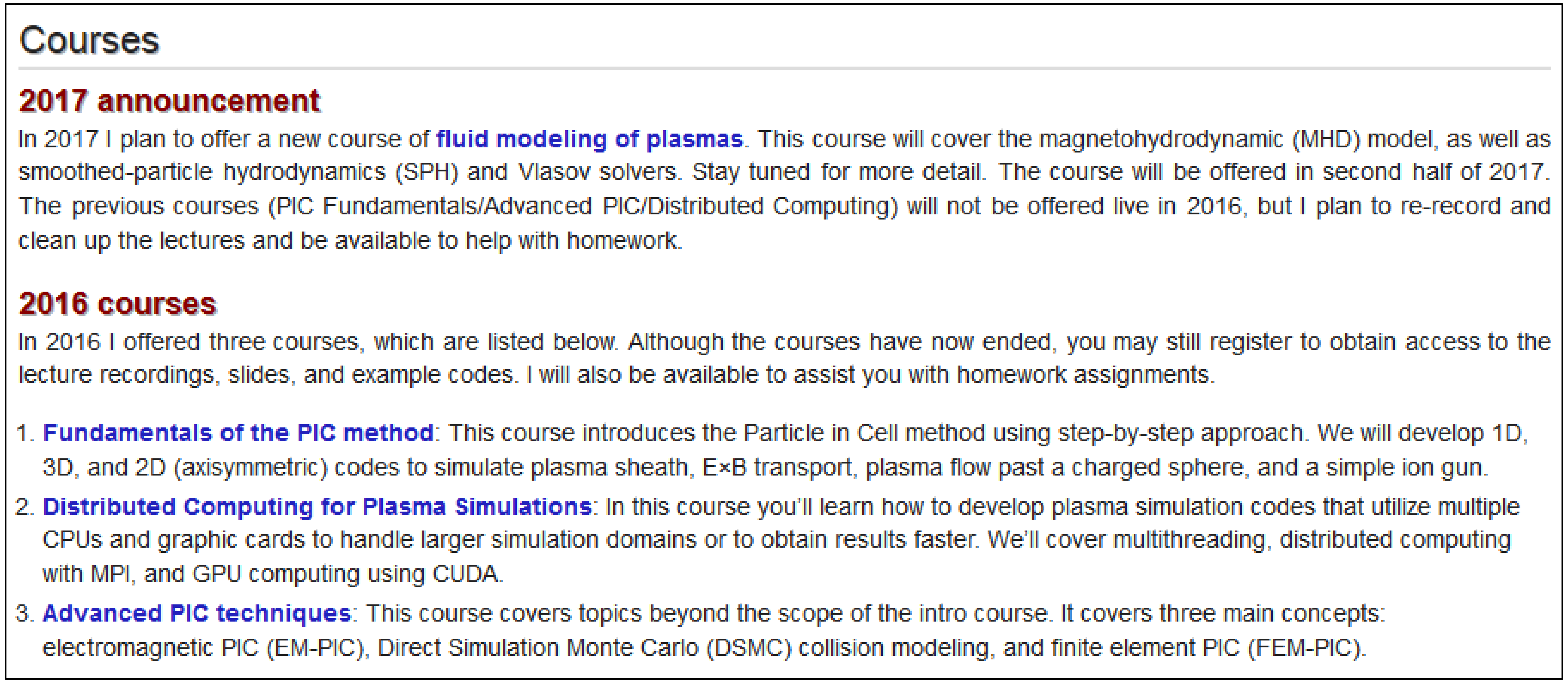
In this webinar, I want to give you the basics needed to start **developing** in Starfish





Courses

- You may also be interested in visiting https://www.particleincell.com/courses to learn more about online courses I offer on various topics related to numerical plasma modeling



Courses

2017 announcement

In 2017 I plan to offer a new course of **fluid modeling of plasmas**. This course will cover the magnetohydrodynamic (MHD) model, as well as smoothed-particle hydrodynamics (SPH) and Vlasov solvers. Stay tuned for more detail. The course will be offered in second half of 2017. The previous courses (PIC Fundamentals/Advanced PIC/Distributed Computing) will not be offered live in 2016, but I plan to re-record and clean up the lectures and be available to help with homework.

2016 courses

In 2016 I offered three courses, which are listed below. Although the courses have now ended, you may still register to obtain access to the lecture recordings, slides, and example codes. I will also be available to assist you with homework assignments.

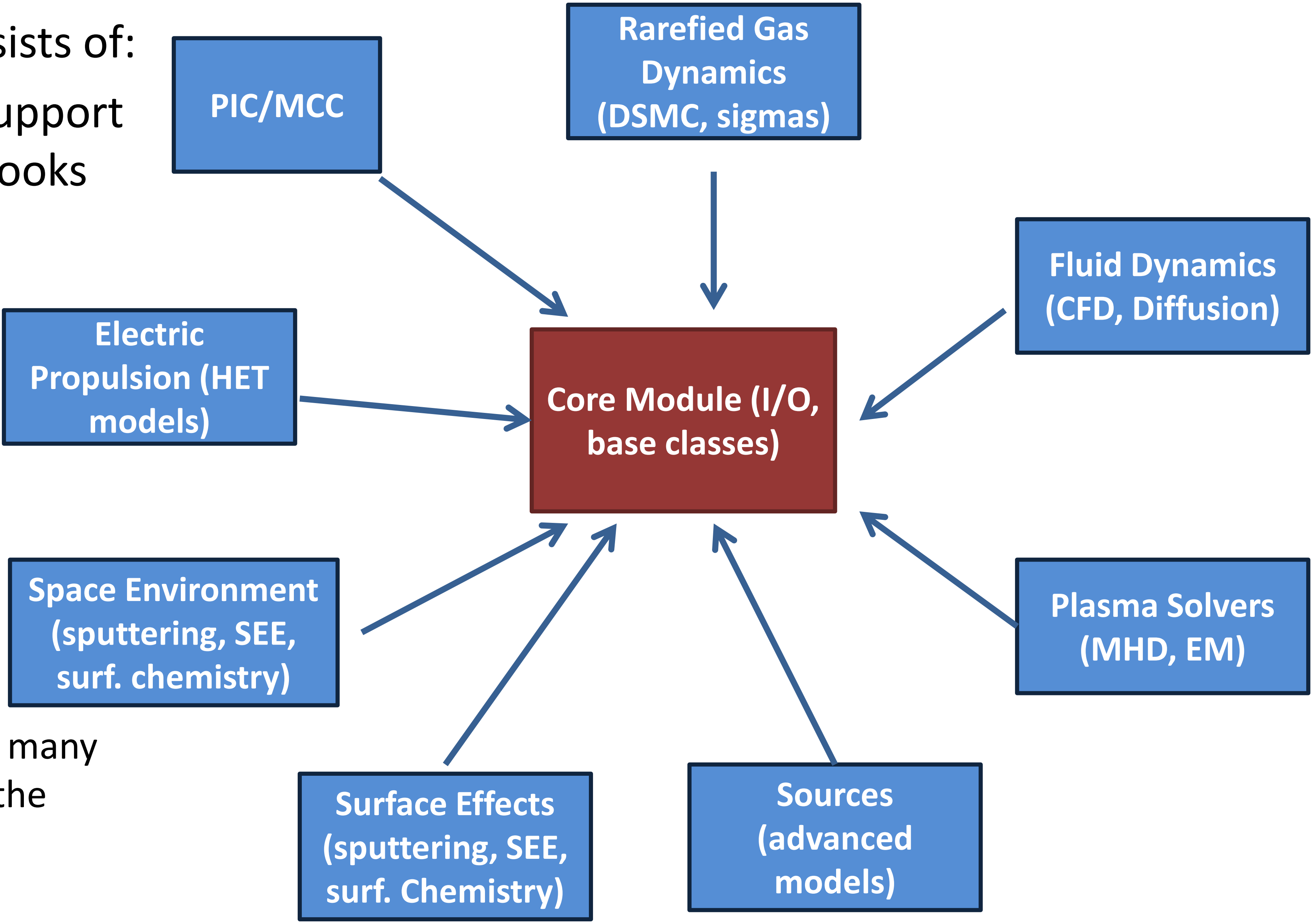
1. **Fundamentals of the PIC method**: This course introduces the Particle in Cell method using step-by-step approach. We will develop 1D, 3D, and 2D (axisymmetric) codes to simulate plasma sheath, E×B transport, plasma flow past a charged sphere, and a simple ion gun.
2. **Distributed Computing for 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 obtain results faster. We'll cover multithreading, distributed computing with MPI, and GPU computing using CUDA.
3. **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).

Code Overview

- Starfish conceptually consists of:

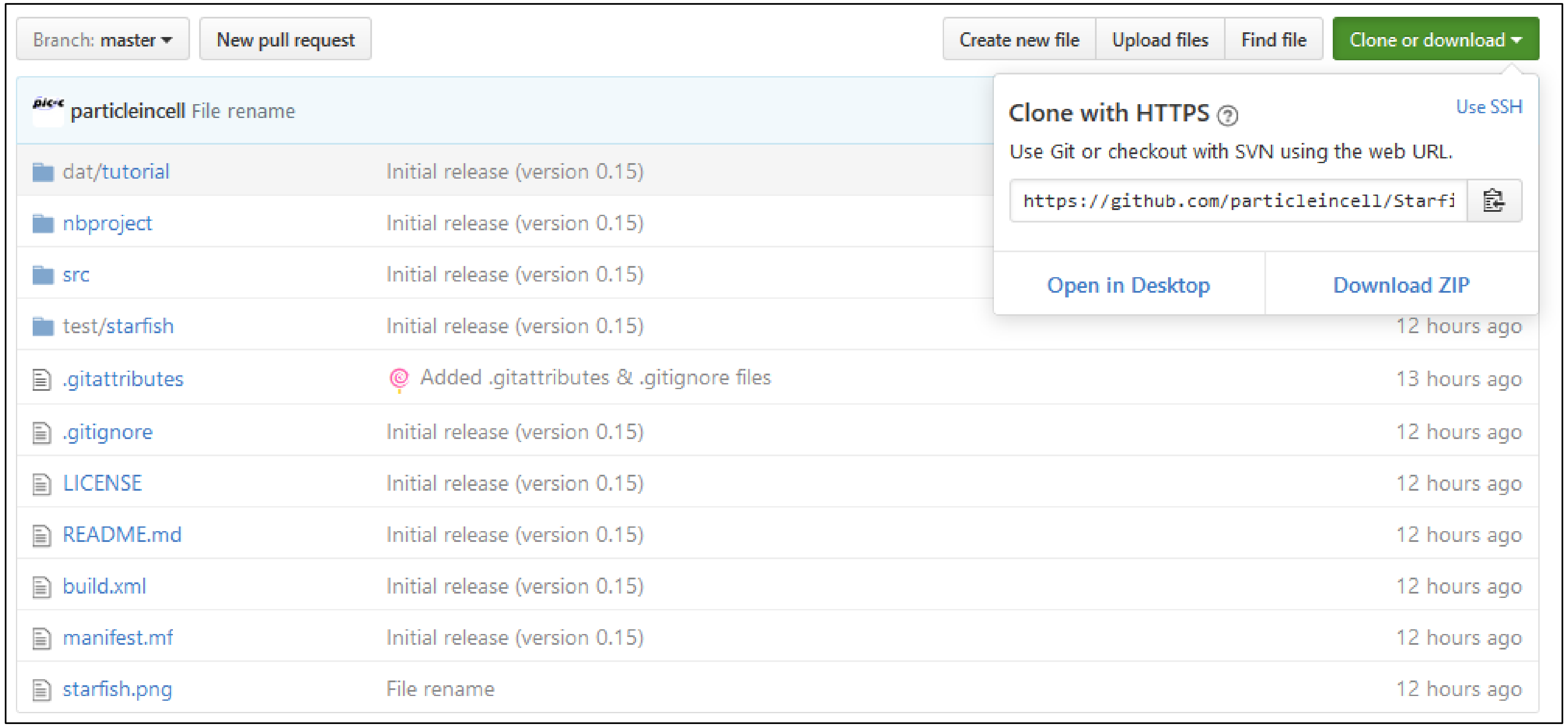
1. **Core** library providing support for I/O and main logic hooks
2. Numerous **modules** implementing relevant physics

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


PIC/MCC
Rarefied Gas Dynamics (DSMC, sigmas)
Fluid Dynamics (CFD, Diffusion)
Electric Propulsion (HET models)
Core Module (I/O, base classes)
Space Environment (sputtering, SEE, surf. chemistry)
Plasma Solvers (MHD, EM)
Surface Effects (sputtering, SEE, surf. Chemistry)
Sources (advanced models)

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

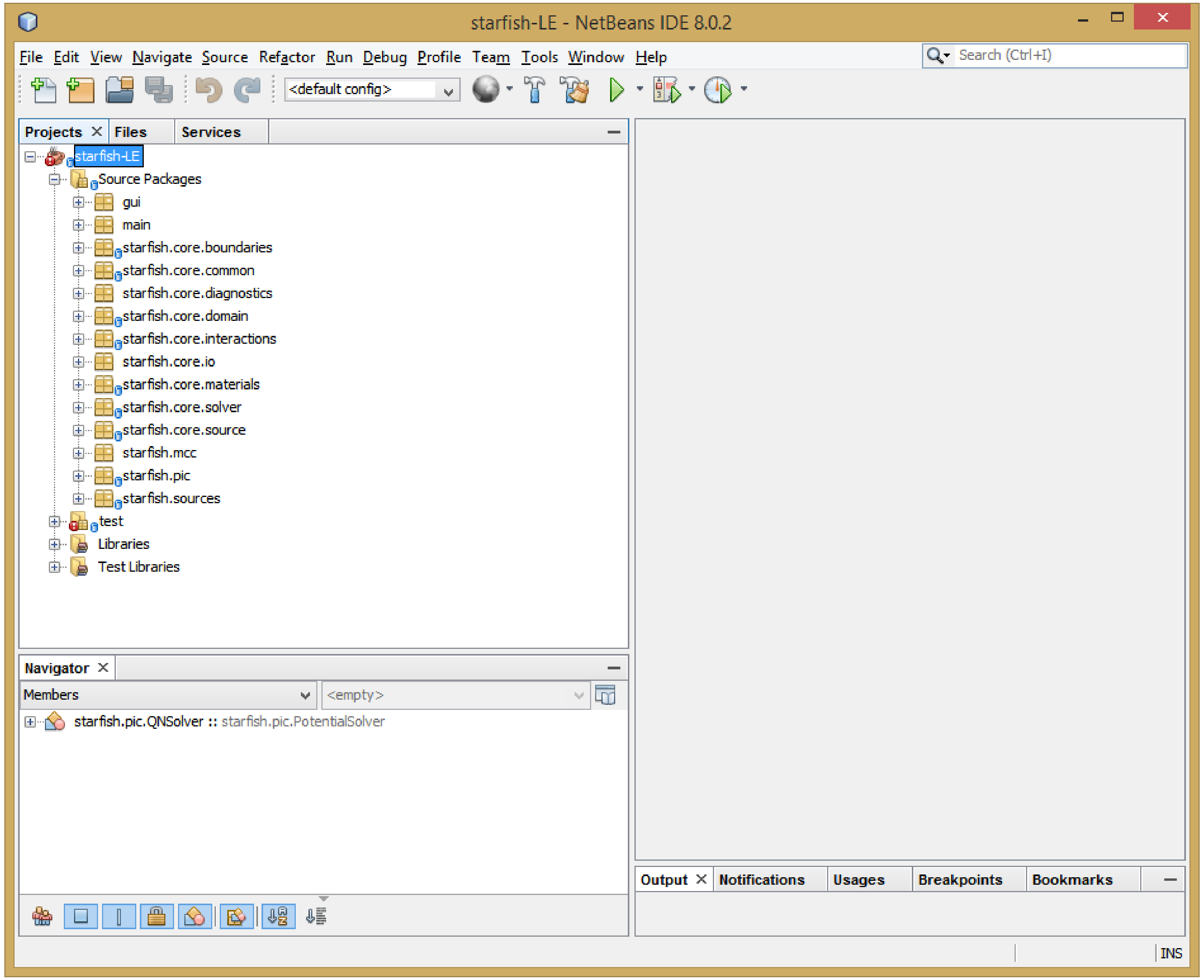
Clone with HTTPS
Use SSH
Use Git or checkout with SVN using the web URL.
https://github.com/particleincell/Starfi
Open in Desktop | Download ZIP

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

– 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

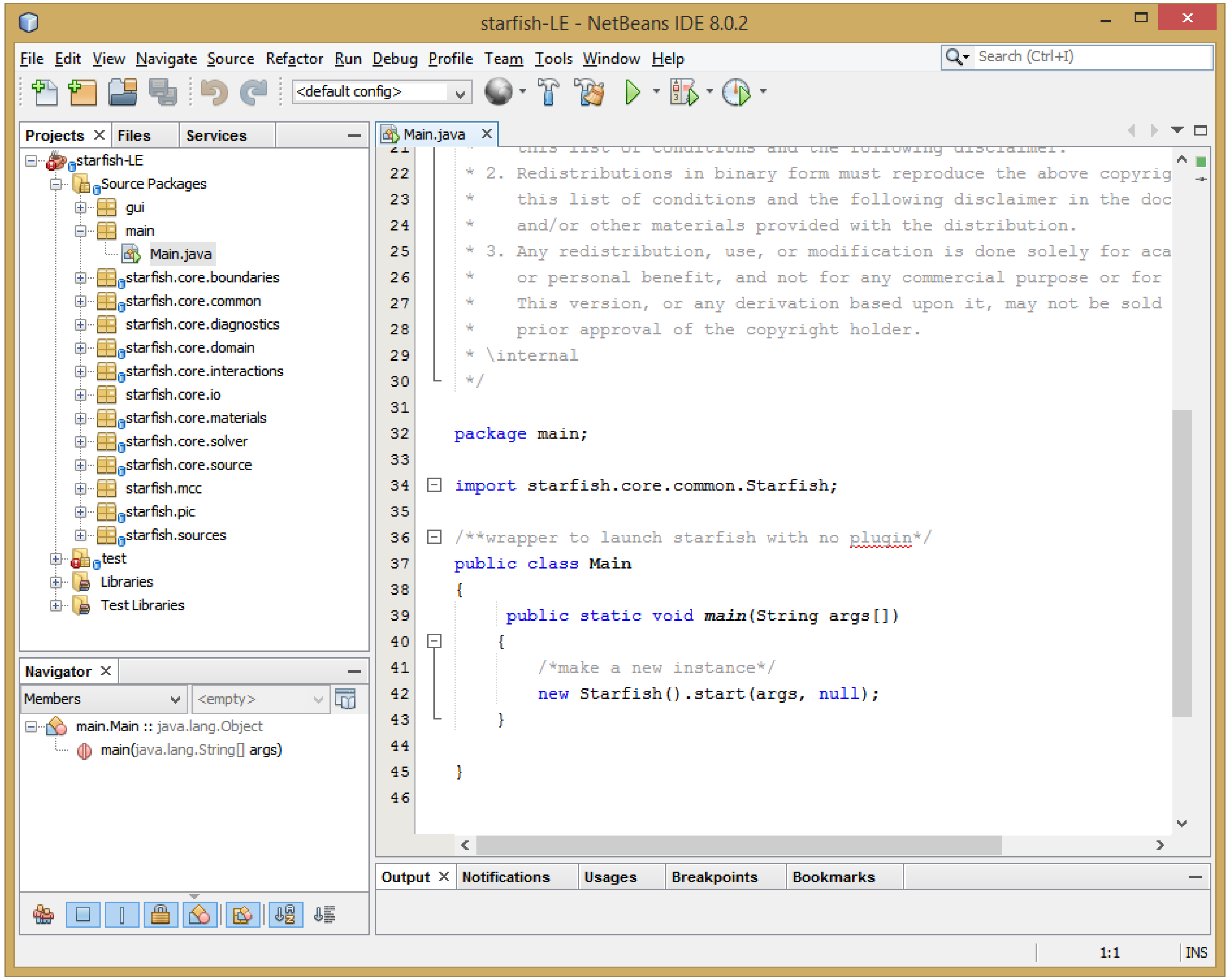

starfish-LE - NetBeans IDE 8.0.2
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Search (Ctrl+I)
<default config>
Projects Files Services
Navigator
Members
<empty>
Output Notifications Usages Breakpoints Bookmarks
INS

```
starfish-LE
  Source Packages
    gui
    main
    starfish.core.boundaries
    starfish.core.common
    starfish.core.diagnostics
    starfish.core.domain
    starfish.core.interactions
    starfish.core.io
    starfish.core.materials
    starfish.core.solver
    starfish.core.source
    starfish.mcc
    starfish.pic
    starfish.sources
  test
  Libraries
  Test Libraries
```

starfish.pic.QNSolver :: starfish.pic.PotentialSolver

Main

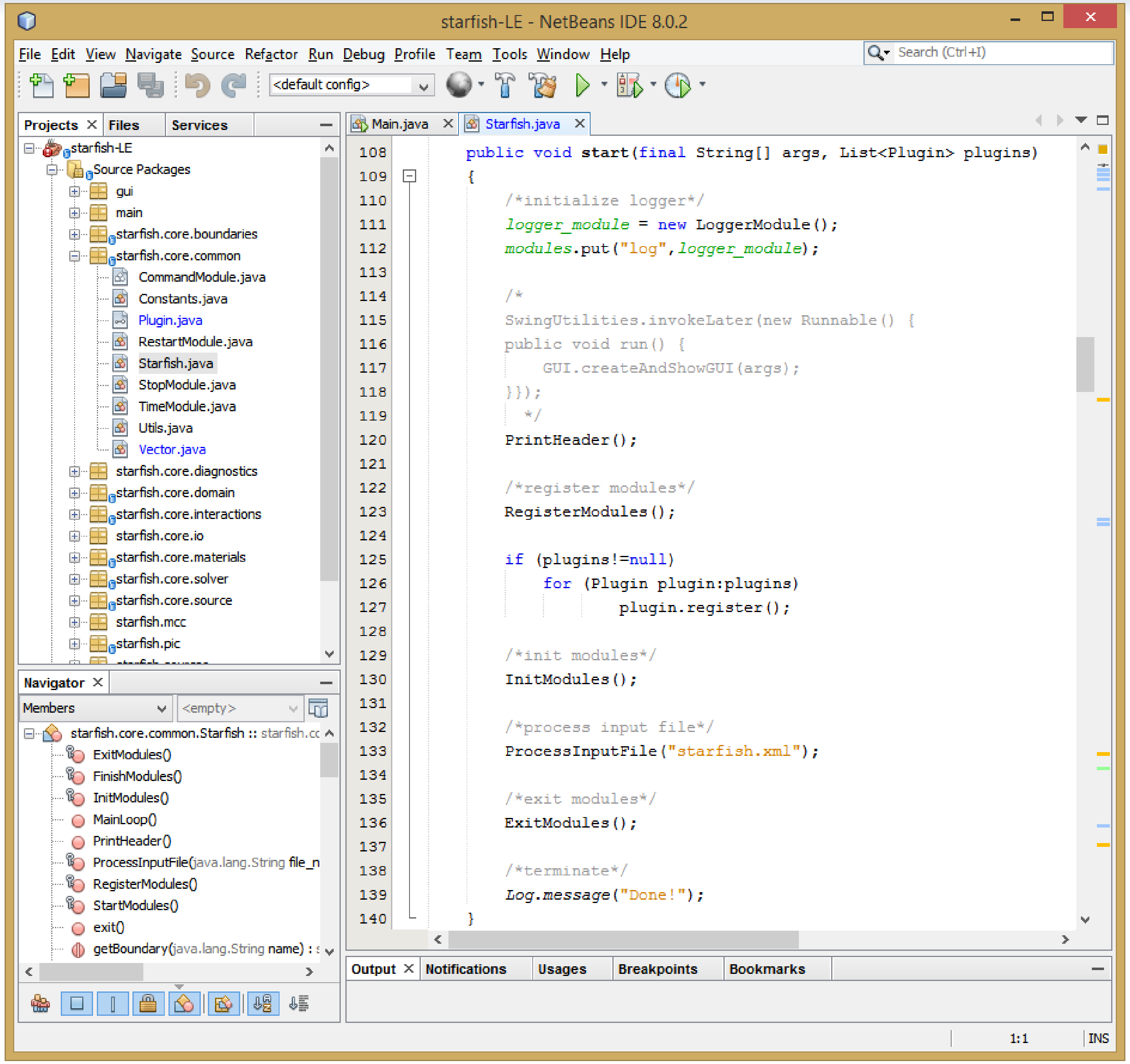
- Just as any Java program, Starfish execution begins in a function called "main"
- This function is defined in Main.java located in package **main**
- This function instantiates a new object of type Starfish and then calls that object's **start** method.



```
 * 2. Redistributions in binary form must reproduce the above copyrig
 *    this list of conditions and the following disclaimer in the doc
 *    and/or other materials provided with the distribution.
 * 3. Any redistribution, use, or modification is done solely for aca
 *    or personal benefit, and not for any commercial purpose or for
 *    This version, or any derivation based upon it, may not be sold
 *    prior approval of the copyright holder.
 * \internal
 */

package main;

import starfish.core.common.Starfish;

/**wrapper to launch starfish with no plugin*/
public class Main
{
    public static void main(String args[])
    {
        /*make a new instance*/
        new Starfish().start(args, null);
    }

}
```

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

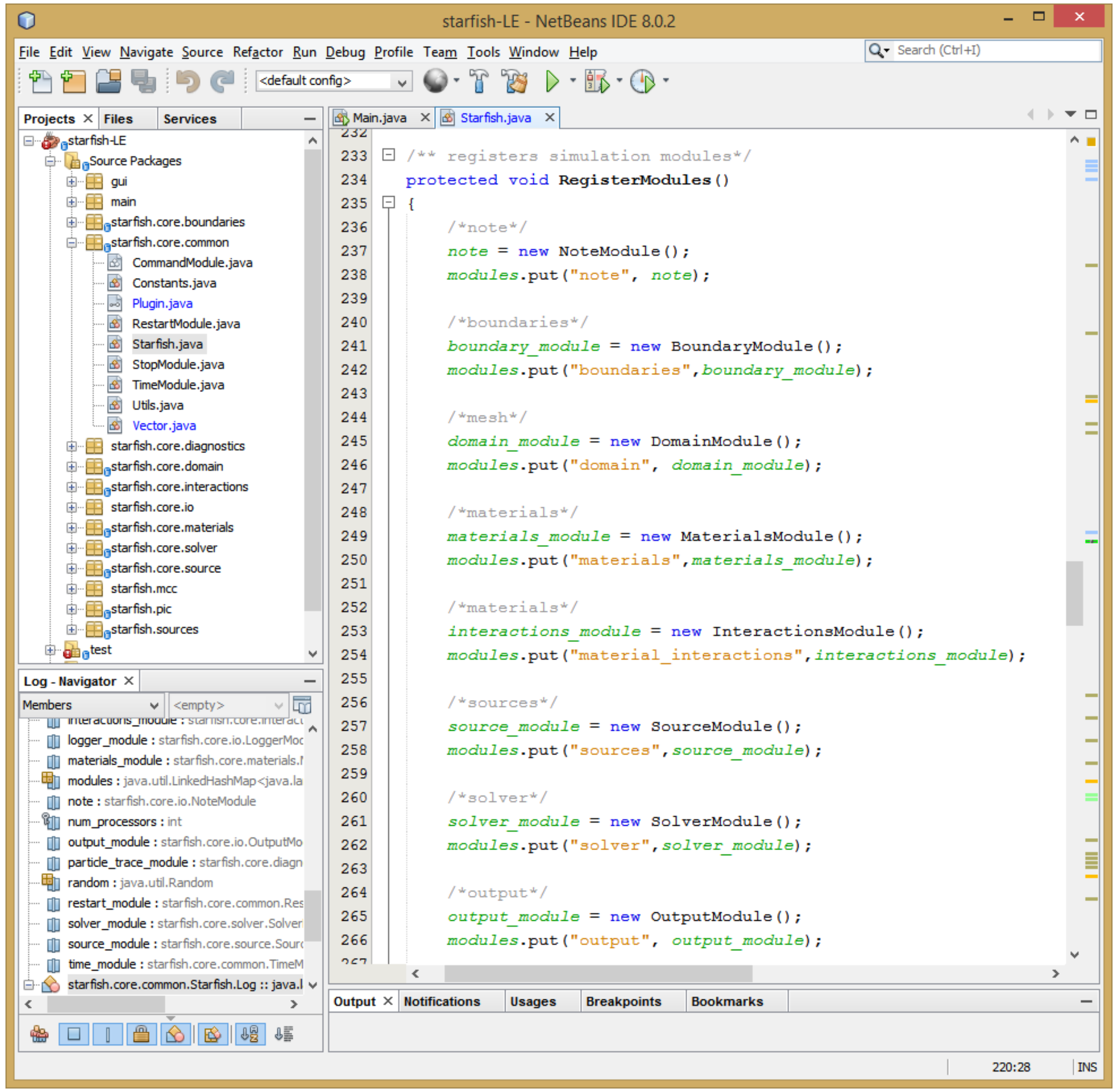


```
public void start(final String[] args, List<Plugin> plugins)
{
    /*initialize logger*/
    logger_module = new LoggerModule();
    modules.put("log",logger_module);

    /*
    SwingUtilities.invokeLater(new Runnable() {
    public void run() {
        GUI.createAndShowGUI(args);
    }});
      */
    PrintHeader();

    /*register modules*/
    RegisterModules();

    if (plugins!=null)
        for (Plugin plugin:plugins)
                plugin.register();

    /*init modules*/
    InitModules();

    /*process input file*/
    ProcessInputFile("starfish.xml");

    /*exit modules*/
    ExitModules();

    /*terminate*/
    Log.message("Done!");
}
```

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



```
interactions_module : starfish.core.interact
logger_module : starfish.core.io.LoggerMoc
materials_module : starfish.core.materials.I
modules : java.util.LinkedHashMap<java.la
note : starfish.core.io.NoteModule
num_processors : int
output_module : starfish.core.io.OutputMo
particle_trace_module : starfish.core.diagn
random : java.util.Random
restart_module : starfish.core.common.Res
solver_module : starfish.core.solver.Solver
source_module : starfish.core.source.Sourc
time_module : starfish.core.common.TimeM
starfish.core.common.Starfish.Log :: java.l
```

```
/** registers simulation modules*/
protected void RegisterModules()
{
    /*note*/
    note = new NoteModule();
    modules.put("note", note);

    /*boundaries*/
    boundary_module = new BoundaryModule();
    modules.put("boundaries",boundary_module);

    /*mesh*/
    domain_module = new DomainModule();
    modules.put("domain", domain_module);

    /*materials*/
    materials_module = new MaterialsModule();
    modules.put("materials",materials_module);

    /*materials*/
    interactions_module = new InteractionsModule();
    modules.put("material_interactions",interactions_module);

    /*sources*/
    source_module = new SourceModule();
    modules.put("sources",source_module);

    /*solver*/
    solver_module = new SolverModule();
    modules.put("solver",solver_module);

    /*output*/
    output_module = new OutputModule();
    modules.put("output", output_module);
```

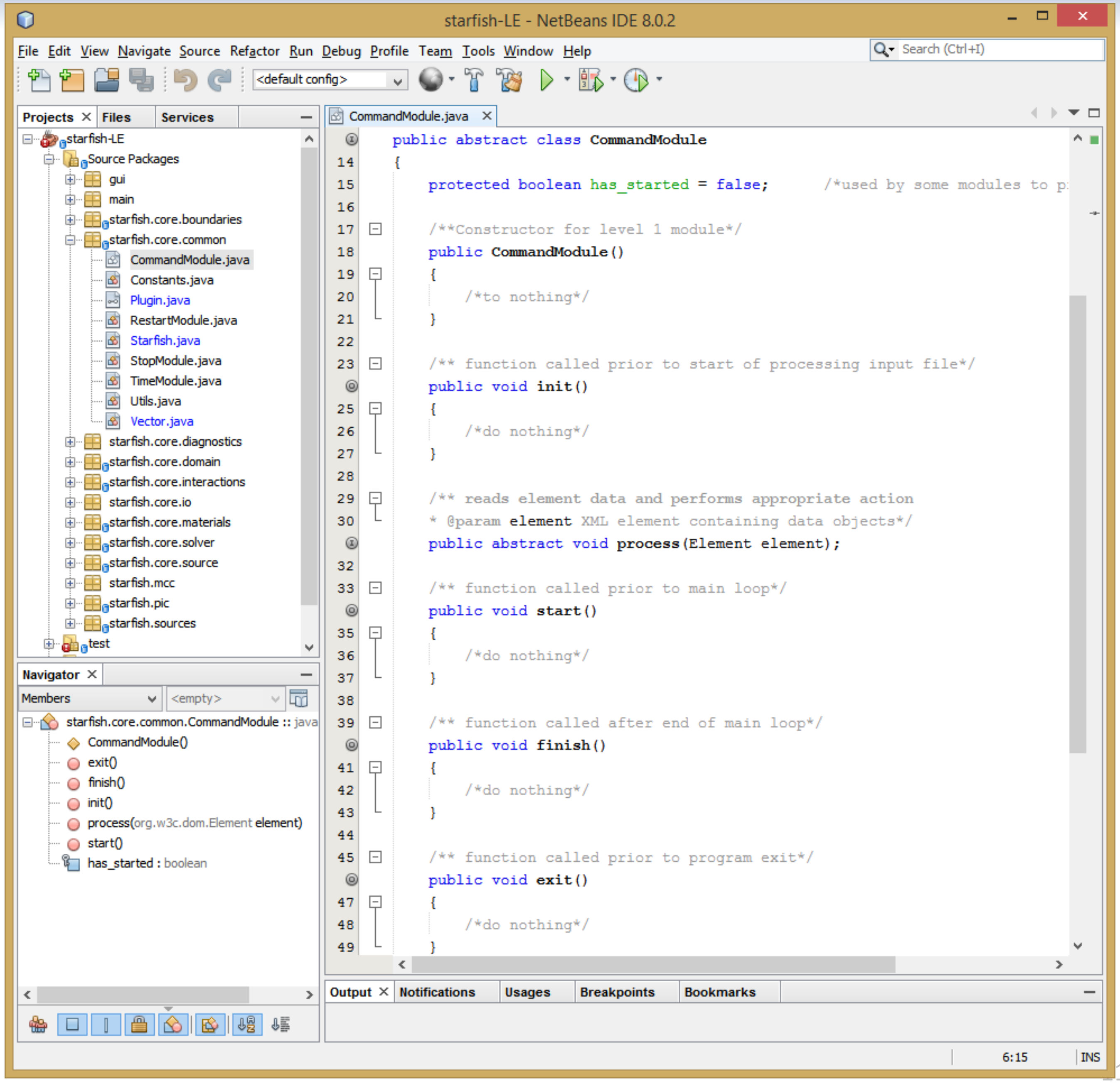
Modules

- Here is the list of currently defined 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_inte-ractions	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

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



```
public abstract class CommandModule
{
    protected boolean has_started = false;      /*used by some modules to p

    /**Constructor for level 1 module*/
    public CommandModule()
    {
        /*to nothing*/
    }

    /** function called prior to start of processing input file*/
    public void init()
    {
        /*do nothing*/
    }

    /** reads element data and performs appropriate action
    * @param element XML element containing data objects*/
    public abstract void process(Element element);

    /** function called prior to main loop*/
    public void start()
    {
        /*do nothing*/
    }

    /** function called after end of main loop*/
    public void finish()
    {
        /*do nothing*/
    }

    /** function called prior to program exit*/
    public void exit()
    {
        /*do nothing*/
    }
```

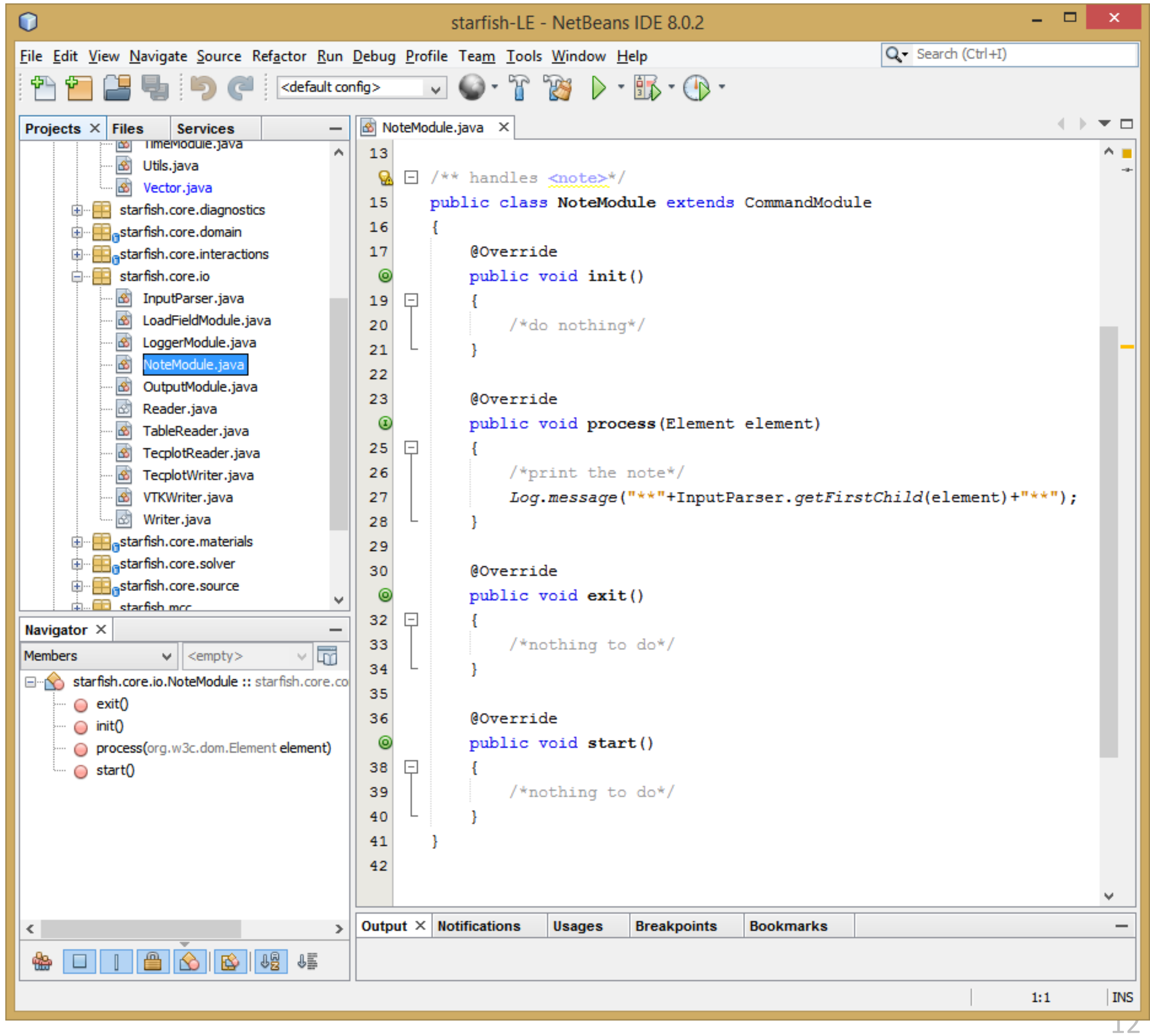
Example: Note Module

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

- 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



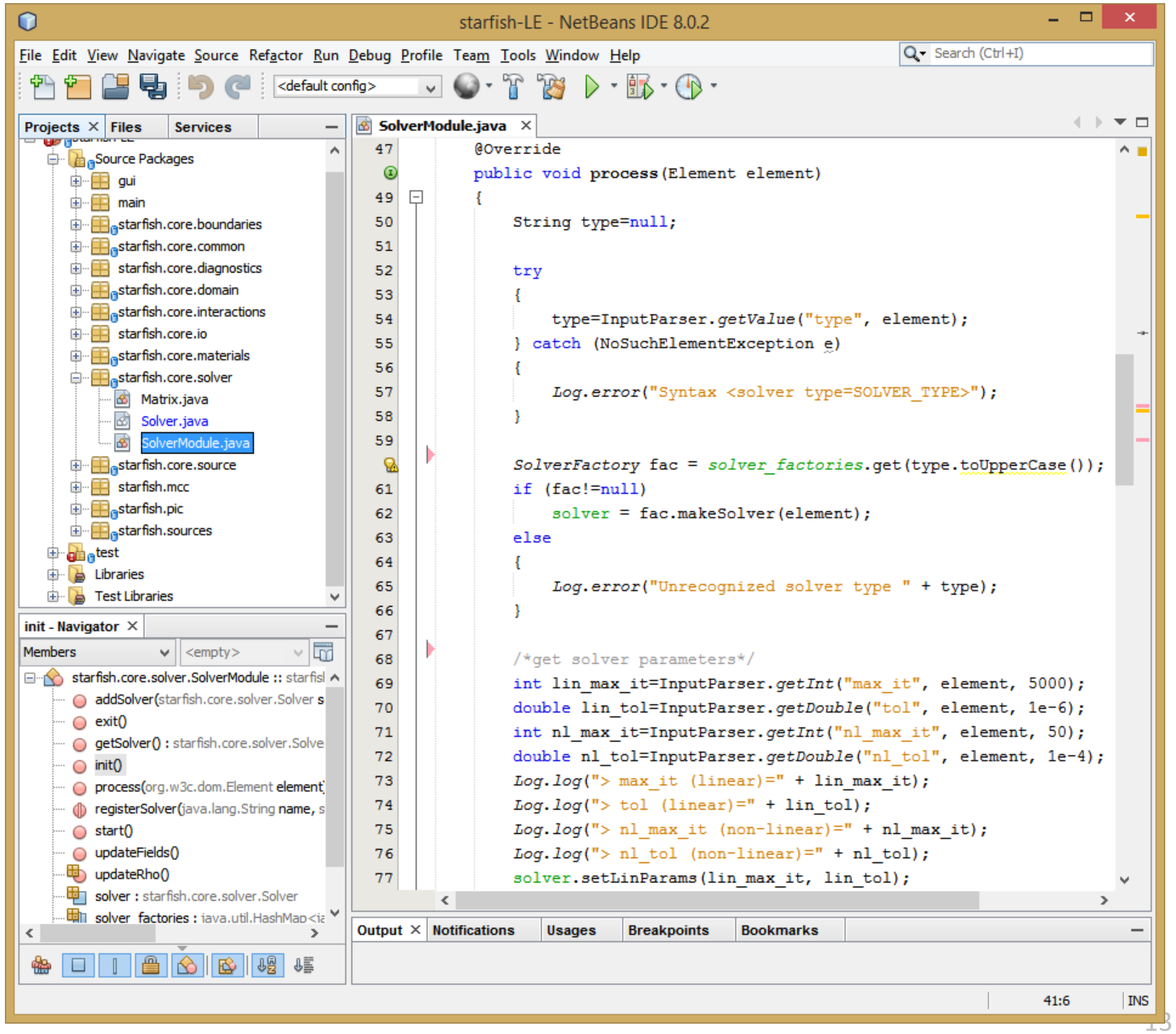
```
/** handles <note>*/
public class NoteModule extends CommandModule
{
    @Override
    public void init()
    {
        /*do nothing*/
    }

    @Override
    public void process(Element element)
    {
        /*print the note*/
        Log.message("**"+InputParser.getFirstChild(element)+"**");
    }

    @Override
    public void exit()
    {
        /*nothing to do*/
    }

    @Override
    public void start()
    {
        /*nothing to do*/
    }
}
```

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>
```

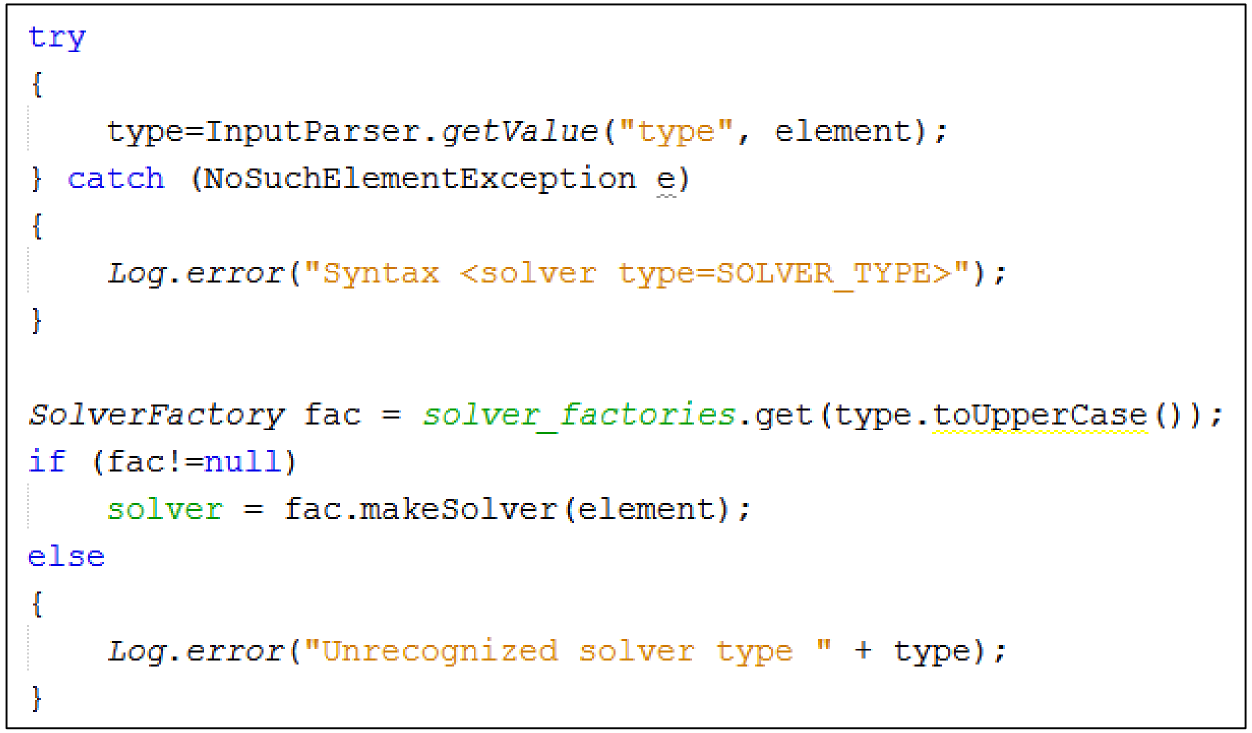
- **InputParser.getValue/getInt/getDouble…** provide easy way to obtain data regardless of whether it was defined as an *attribute* (type) or *child elements* (n0, Te0…)


starfish-LE - NetBeans IDE 8.0.2
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Search (Ctrl+I)
<default config>
Projects Files Services
Source Packages
gui
main
starfish.core.boundaries
starfish.core.common
starfish.core.diagnostics
starfish.core.domain
starfish.core.interactions
starfish.core.io
starfish.core.materials
starfish.core.solver
Matrix.java
Solver.java
SolverModule.java
starfish.core.source
starfish.mcc
starfish.pic
starfish.sources
test
Libraries
Test Libraries
init - Navigator
Members
<empty>
starfish.core.solver.SolverModule :: starfis
addSolver(starfish.core.solver.Solver s
exit()
getSolver() : starfish.core.solver.Solve
init()
process(org.w3c.dom.Element element)
registerSolver(java.lang.String name, s
start()
updateFields()
updateRho()
solver : starfish.core.solver.Solver
solver_factories : java.util.HashMap<
SolverModule.java
Output Notifications Usages Breakpoints Bookmarks
41:6
INS

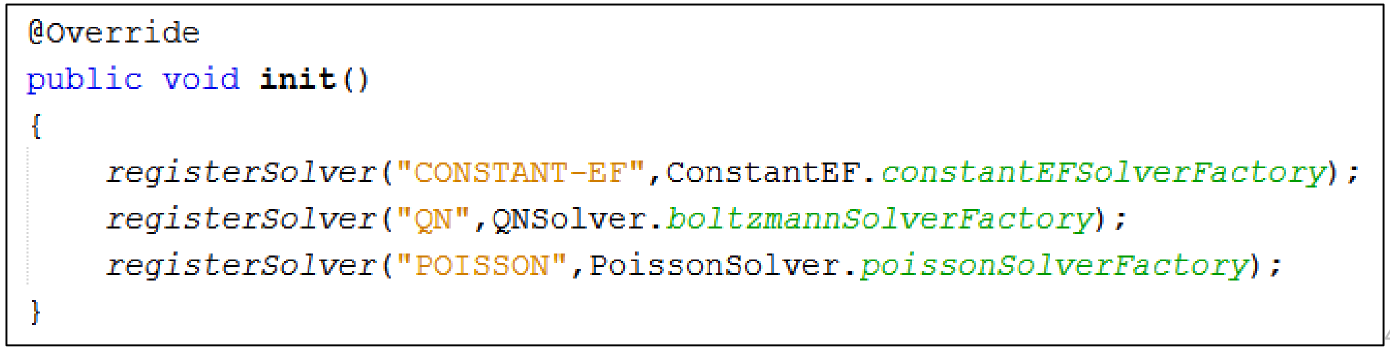
```java
    @Override
    public void process(Element element)
    {
        String type=null;

        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);
        }

        /*get solver parameters*/
        int lin_max_it=InputParser.getInt("max_it", element, 5000);
        double lin_tol=InputParser.getDouble("tol", element, 1e-6);
        int nl_max_it=InputParser.getInt("nl_max_it", element, 50);
        double nl_tol=InputParser.getDouble("nl_tol", element, 1e-4);
        Log.log("> max_it (linear)=" + lin_max_it);
        Log.log("> tol (linear)=" + lin_tol);
        Log.log("> nl_max_it (non-linear)=" + nl_max_it);
        Log.log("> nl_tol (non-linear)=" + nl_tol);
        solver.setLinParams(lin_max_it, lin_tol);
```

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



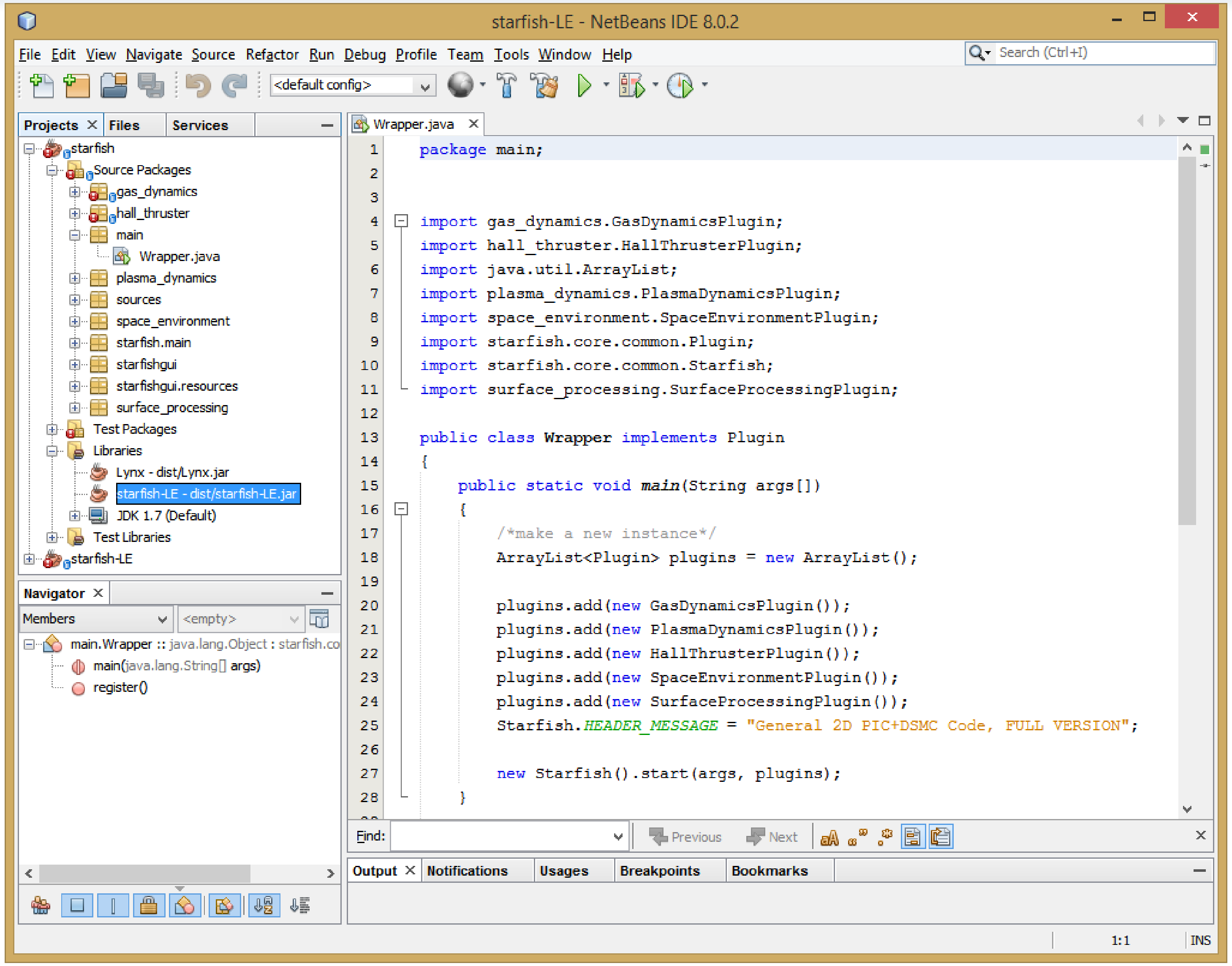
```
try
{
    type=InputParser.getValue("type", element);
} catch (NoSuchElementException e)
{
    Log.error("Syntax <solver type=SOLVER_TYPE>");
}

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**



```java
package main;


import gas_dynamics.GasDynamicsPlugin;
import hall_thruster.HallThrusterPlugin;
import java.util.ArrayList;
import plasma_dynamics.PlasmaDynamicsPlugin;
import space_environment.SpaceEnvironmentPlugin;
import starfish.core.common.Plugin;
import starfish.core.common.Starfish;
import surface_processing.SurfaceProcessingPlugin;

public class Wrapper implements Plugin
{
    public static void main(String args[])
    {
        /*make a new instance*/
        ArrayList<Plugin> plugins = new ArrayList();

        plugins.add(new GasDynamicsPlugin());
        plugins.add(new PlasmaDynamicsPlugin());
        plugins.add(new HallThrusterPlugin());
        plugins.add(new SpaceEnvironmentPlugin());
        plugins.add(new SurfaceProcessingPlugin());
        Starfish.HEADER_MESSAGE = "General 2D PIC+DSMC Code, FULL VERSION";

        new Starfish().start(args, plugins);
    }
```

- **Register** method of each plugin will be called after **RegisterModules**
  - Your plugin could for instance call

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

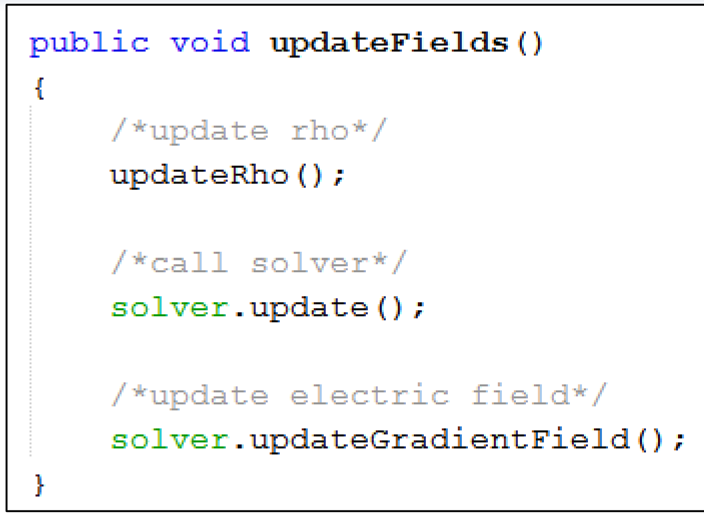
Plugins

- 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

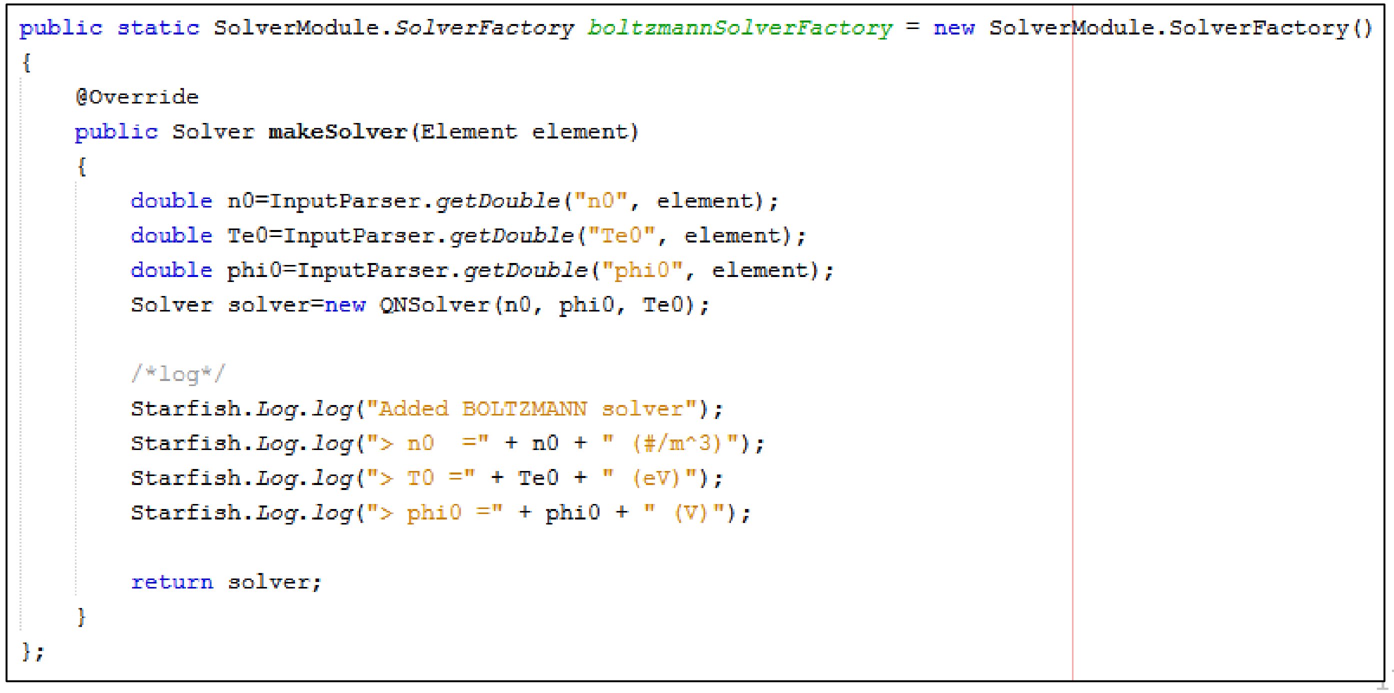
```
package gas_dynamics;

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



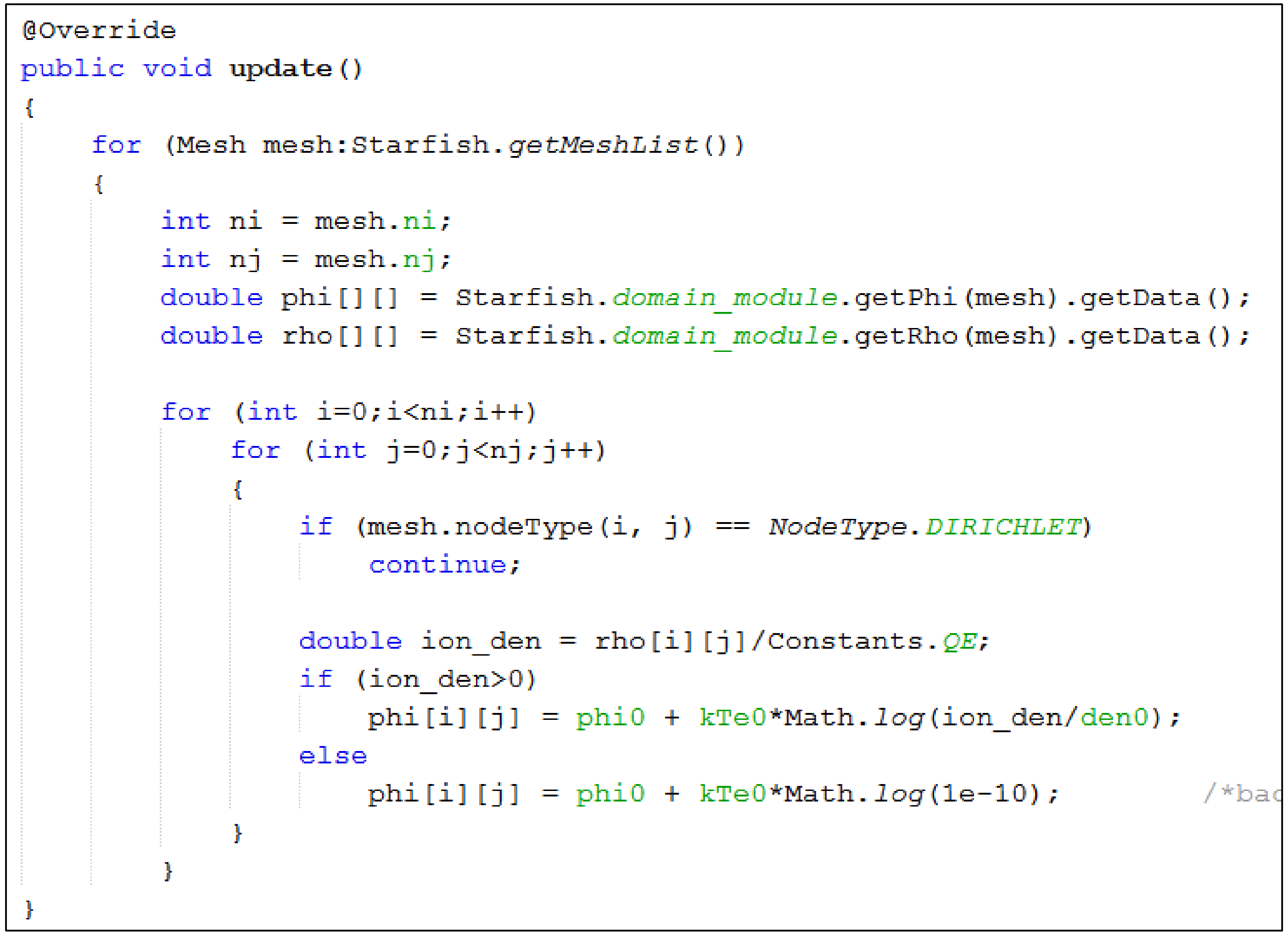
```
public void updateFields()
{
    /*update rho*/
    updateRho();

    /*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

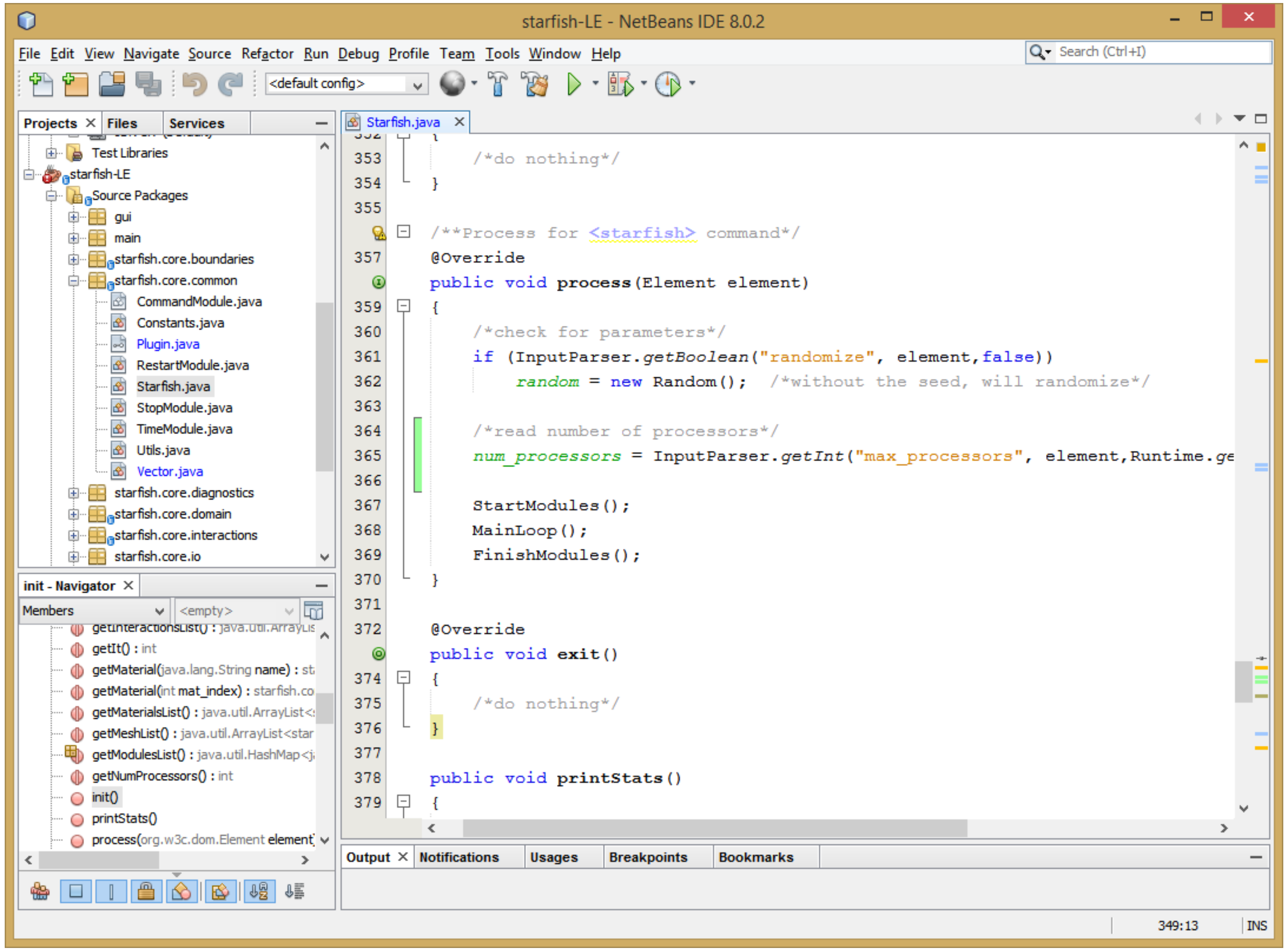
- Starfish stores mesh-based quantities in object of type Field2D. This object defines functions for interpolation and also getData which returns double[][] containing the actual node values.
- The domain module automatically generates fields to store charge density ρ and potential ϕ
- Each mesh node 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);        /*bac
            }
        }
}
```

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


starfish-LE - NetBeans IDE 8.0.2
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Search (Ctrl+I)
<default config>
Projects Files Services
Test Libraries
starfish-LE
Source Packages
gui
main
starfish.core.boundaries
starfish.core.common
CommandModule.java
Constants.java
Plugin.java
RestartModule.java
Starfish.java
StopModule.java
TimeModule.java
Utils.java
Vector.java
starfish.core.diagnostics
starfish.core.domain
starfish.core.interactions
starfish.core.io
init - Navigator
Members
<empty>
getInteractionsList() : java.util.ArrayLis
getIt() : int
getMaterial(java.lang.String name) : st
getMaterial(int mat_index) : starfish.co
getMaterialsList() : java.util.ArrayList<
getMeshList() : java.util.ArrayList<star
getModulesList() : java.util.HashMap<j
getNumProcessors() : int
init()
printStats()
process(org.w3c.dom.Element element)
Starfish.java
Output Notifications Usages Breakpoints Bookmarks
349:13
INS

```
        /*do nothing*/
    }

    /**Process for <starfish> command*/
    @Override
    public void process(Element element)
    {
        /*check for parameters*/
        if (InputParser.getBoolean("randomize", element,false))
            random = new Random();  /*without the seed, will randomize*/

        /*read number of processors*/
        num_processors = InputParser.getInt("max_processors", element,Runtime.ge

        StartModules();
        MainLoop();
        FinishModules();
    }

    @Override
    public void exit()
    {
        /*do nothing*/
    }

    public void printStats()
    {
```

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

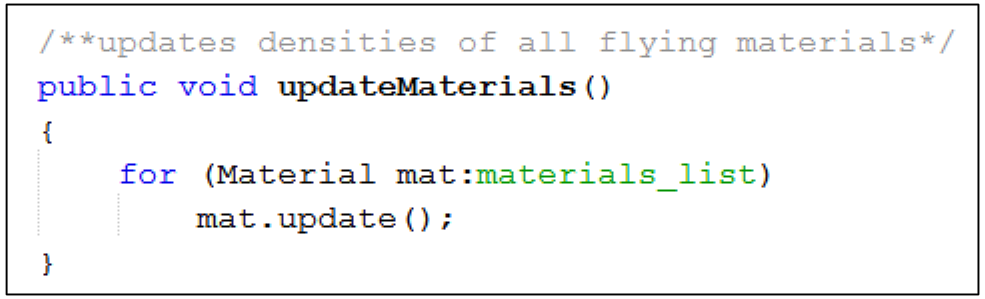
- This is what it looks like in practice

```
 /**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

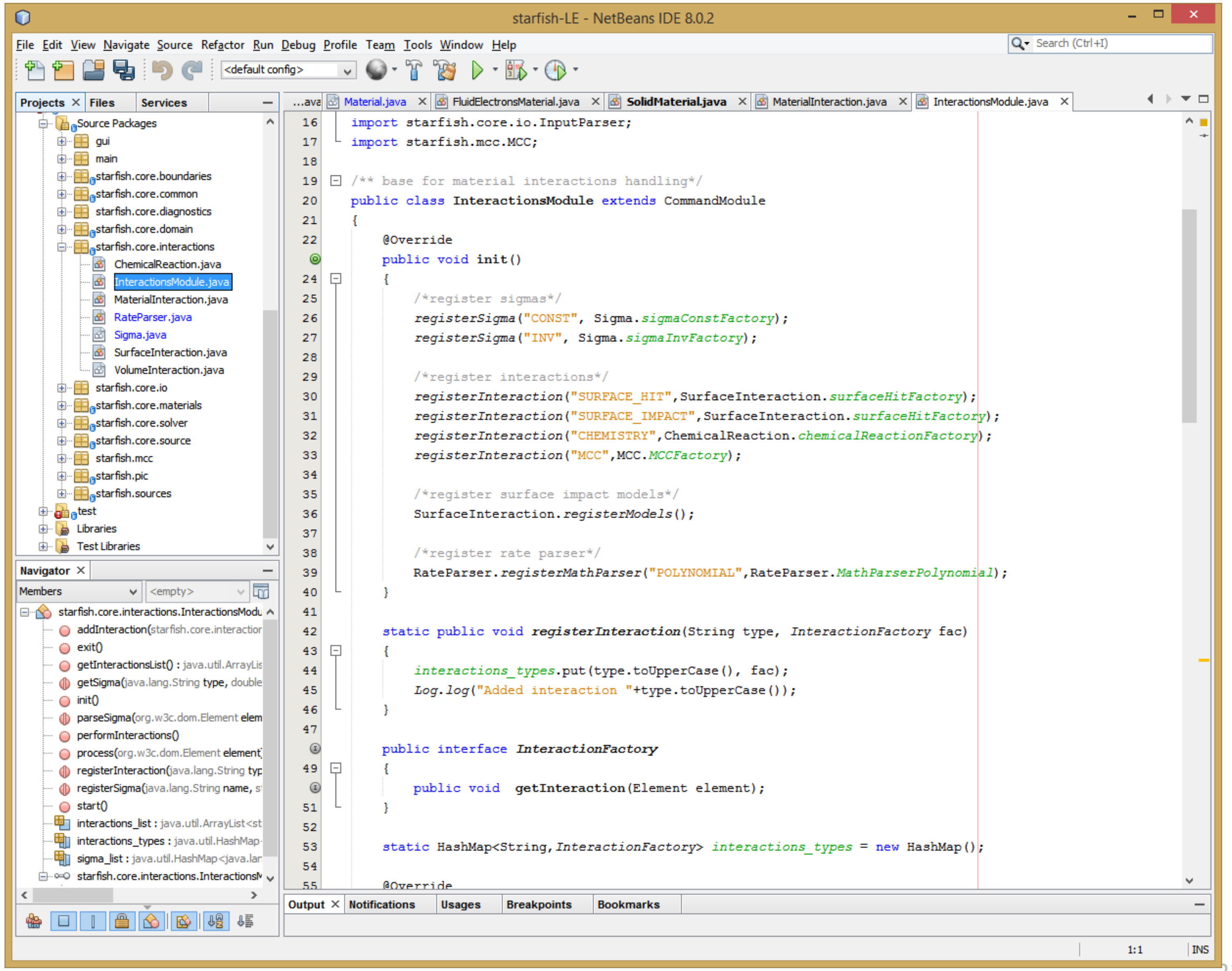
- The **updateMaterials** function is another example of modularity afforded by object oriented programming
- While Starfish is mainly used for kinetic simulations, the code is not hardwired for this
- Even in a PIC simulation, we merely care about the **density** (and possibly velocities/temperature) 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.

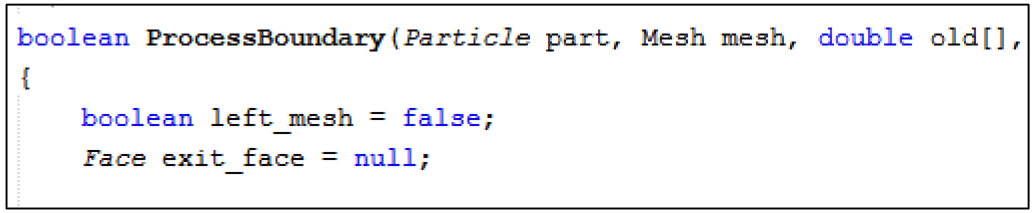


```java
import starfish.core.io.InputParser;
import starfish.mcc.MCC;

/** base for material interactions handling*/
public class InteractionsModule extends CommandModule
{
    @Override
    public void init()
    {
        /*register sigmas*/
        registerSigma("CONST", Sigma.sigmaConstFactory);
        registerSigma("INV", Sigma.sigmaInvFactory);

        /*register interactions*/
        registerInteraction("SURFACE_HIT",SurfaceInteraction.surfaceHitFactory);
        registerInteraction("SURFACE_IMPACT",SurfaceInteraction.surfaceHitFactory);
        registerInteraction("CHEMISTRY",ChemicalReaction.chemicalReactionFactory);
        registerInteraction("MCC",MCC.MCCFactory);

        /*register surface impact models*/
        SurfaceInteraction.registerModels();

        /*register rate parser*/
        RateParser.registerMathParser("POLYNOMIAL",RateParser.MathParserPolynomial);
    }

    static public void registerInteraction(String type, InteractionFactory fac)
    {
        interactions_types.put(type.toUpperCase(), fac);
        Log.log("Added interaction "+type.toUpperCase());
    }

    public interface InteractionFactory
    {
        public void  getInteraction(Element element);
    }

    static HashMap<String,InteractionFactory> interactions_types = new HashMap();

    @Override
```

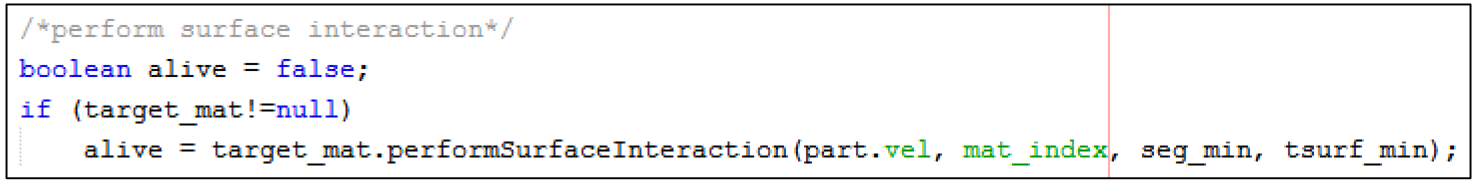
```
starfish.core.interactions.InteractionsModu
addInteraction(starfish.core.interaction
exit()
getInteractionsList() : java.util.ArrayLis
getSigma(java.lang.String type, double
init()
parseSigma(org.w3c.dom.Element elem
performInteractions()
process(org.w3c.dom.Element element)
registerInteraction(java.lang.String typ
registerSigma(java.lang.String name, s
start()
interactions_list : java.util.ArrayList<st
interactions_types : java.util.HashMap
sigma_list : java.util.HashMap<java.lar
starfish.core.interactions.InteractionsM
```

Surface Interactions

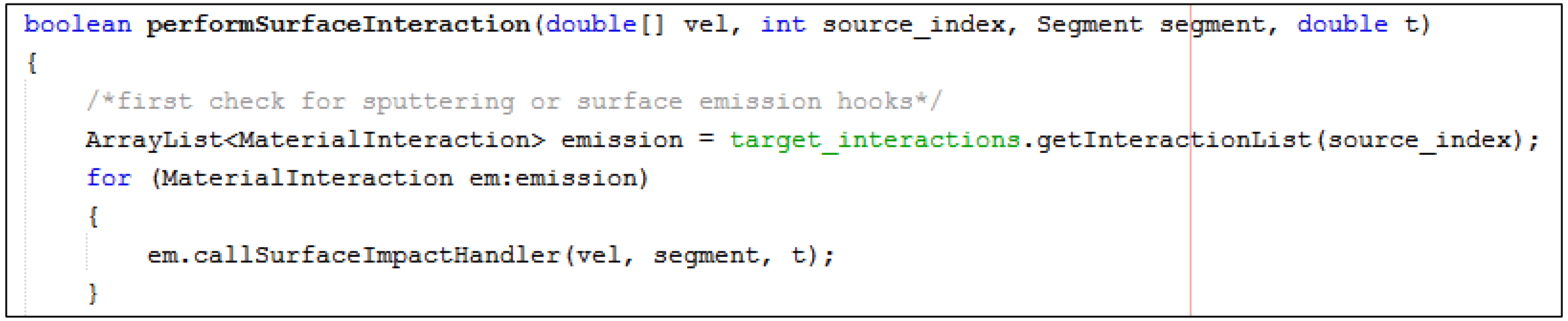
- The division on the previous page is actually bit of a simplification. Material Interactions are actually grouped into **surface interactions** and **volume interactions**. MCC, DSMC, and Chemistry are subclasses of VolumeInteraction since they deal with effects within the computational mesh (volume).
- 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.perfromInteraction. 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();
}
```

```
public void perform()
{
    Log.debug("performing MCC");
    for (Mesh mesh:Starfish.getMeshList())
    {
        Iterator<Particle> iterator = source.getIterator(mesh);
        Field2D target_den = target.getDen(mesh);

        while (iterator.hasNext())
        {
            Particle part = iterator.next();

            double den_a = target_den.gather(part.lc);

            /*create random target particle*/
            double target_vel[] = target.sampleVelocity(mesh,part.lc);

            double g = Vector.mag2(part.vel);

            /*collision probability*/
            /*TODO: implement multiple interactions*/
            double P = 1-Math.exp(-sigma.eval(g)*g*Starfish.getDt()*den_a);
```

Conclusion

- That is all for today
- 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