Análisis teórico del contacto plasma superficie y sus aplicaciones industriales

Memoria presentada para optar al grado de:

Doctor en Física

por:

Tejero Del Caz, Antonio

En Córdoba a, 13 de mayo de 2016

TITULO: *Análisis teórico del contacto plasma-superficie y sus aplicaciones industriales*

AUTOR: *Antonio Tejero del Caz*

© Edita: Servicio de Publicaciones de la Universidad de Córdoba. 2016
Campus de Rabanales
Ctra. Nacional IV, Km. 396 A
14071 Córdoba

www.uco.es/publicaciones
publicaciones@uco.es

TÍTULO DE LA TESIS: **Análisis teórico del contacto plasma superficie y sus aplicaciones industriales**

DOCTORANDO/A: **Antonio Tejero del Caz**

INFORME RAZONADO DEL/DE LOS DIRECTOR/ES DE LA TESIS

(se hará mención a la evolución y desarrollo de la tesis, así como a trabajos y publicaciones derivados de la misma).

El tema de la tesis doctoral que se va a presentar ha sido el desarrollo de simulaciones Paticle in Cell para estudiar el contacto de una superficie metálica con un plasma neutro en diversas condiciones. La forma de abordar el problema ha sido muy original desde el principio, haciendo uso de tarjetas gráficas (GPUs) y programación en paralelo para desarrollar las simulaciones. El doctorando ha sido muy autónomo en el aprendizaje de este tipo de tarjetas y de su programación. Las simulaciones desarrolladas han sido mejoradas conforme se iba avanzando en la investigación, centrándose en aquellas que han proporcionando resultados originales y que no se pueden obtener con los modelos clásicos de fluidos. Uno de los mayores logros ha sido el desarrollo de una simulación del contacto de un plasma con una sonda de Langmuir cilíndrica que ha permitido analizar la transición desde el comportamiento radial del movimiento de los iones positivos, presentes en el plasma, hacia la sonda hasta el comportamiento orbital. Se ha demostrado que el parámetro determinante en esta transición es la razón entre la temperatura de iones iones y la electrónica. Lo más novedoso y que ha permitido estudiar esta transición ha sido el modo de alimentar la simulación, considerando una función de distribución de los iones positivos en la entrada de la simulación consistente en una distribución maxwelliana desplazada en velocidades. El desplazamiento en la velocidad es una consecuencia de todo lo que ocurre desde el plasma imperturbado hasta la entrada de la simulación. Esta forma de alimentar la simulación resuelve el problema conocido como "vaina de la fuente" encontrado por el resto de investigadores que abordan problemas similares.

Los resultados obtenidos son muy relevantes, lo que le ha permitido publicar una primera parte de estos resultados en un letter, en la revista Plasma Sources, Science and Technology, que se encuentra la número 2 de 31 en el área Physics, Fluids & Plasmas del Journal Citation Report (Tejero del Caz A., Fernández Palop J. I., Díaz Cabrera J. M. & Ballesteros J., *Radial to orbital motion transition in cylindrical Langmuir probes studied with Particle-in-Cell simulations*, (2016) Plasma Sources Sciences & Technology **25** 01LT03).

Además de la publicación mencionada en el párrafo anterior, el doctorando ha participado, no sólo en su investigación sino en la que desarrollan el resto de miembros del equipo investigador, lo que le ha permitido ser co-autor de 5 artículos publicados en revistas destacadas del Journal Citation Report. También ha participado en 12 comunicaciones presentadas en congresos nacionales e internacionales. Ha realizado una estancia de 3 meses con el profesor Tomaž Gyergyek de la Facultad de

ingeniería eléctrica de la Universidad de Ljubljana (Eslovenia), para la que recibió una ayuda del Ministerio de Educación, Cultura y Deporte y que le permitirá obtener el grado de Doctor con mención Internacional. Finalmente, ha participado como asistente en diversos congresos internacionales, el último de ellos, el Internacional Conference on Phenomena in Ionizad Gases celebrado en Iasi (Rumanía).

Por todo ello, se autoriza la presentación de la tesis doctoral.

Córdoba, __11__ de __mayo____ de _2016__

Firma del/de los director/es

Fdo.:_José Ignacio Fernández Palop__ Fdo.: ________________

Contents

Agradecimientos/Acknowledgements 1

Resumen/Abstract 3

I Theoretical Foundations 9

1. Plasmas, sheaths, probes and diagnosis 11

1.1. Introduction . . . 11
1.2. What is a Plasma? . . . 11
1.3. Debye shielding . . . 13
1.4. Contact of a plasma and a metallic surface . . . 16
1.4.1. Quasineutral solution . . . 19
1.4.2. Sheath solution . . . 20
1.4.3. Complete solution . . . 23
1.5. Langmuir probes and plasma diagnosis . . . 24
1.5.1. Different probe types . . . 24
1.5.2. Current to voltage characteristic curve of a Langmuir probe . . . 25
1.6. Conclusion . . . 27

2. Theories of the ion current collected by a cylindrical probe 29

2.1. Introduction . . . 29
2.2. Ion saturation zone and cylindrical probes. Why? . . . 29
2.3. Orbital theories of the ion current collected by a cylindrical probe . . . 31
2.3.1. Mott-Smith and Langmuir model . . . 32
2.3.2. Bernstein and Rabinowitz model . . . 36
2.3.3. Laframboise model . . . 41
2.4. Radial theories of the ion current collected by a cylindrical probe . . . 42
2.4.1. Allen, Boyd and Reynolds / Chen model (ABR model) . . . 42
2.4.2. Fernández Palop model . . . 45
2.4.3. Morales Crespo model . . . 50
2.5. Comparison between orbital and radial theories . . . 53
2.5.1. Sonin-plot . . . 54

2.6. Conclusion . . . 56

II Particle Simulations **57**

3. Particle-In-Cell simulations & parallelisation techniques **59**

3.1. Introduction . . . 59

3.2. Particle simulations and computer experiments . . . 60

3.3. The Particle-In-Cell method . . . 64

3.3.1. Force evaluation . . . 65

3.3.2. Integration of the equations of motion . . . 70

3.3.3. Boundary conditions . . . 72

3.4. Need of parallelism and the GPGPU approach . . . 74

3.5. The CUDA® framework . . . 77

3.5.1. Thread hierarchy . . . 78

3.5.2. Thread synchronisation and memory hierarchy . . . 80

3.5.3. Heterogeneous programming model . . . 81

3.6. Conclusion . . . 83

4. PIC simulation of a planar Langmuir probe (CUPIC1D1V_PP) **85**

4.1. Introduction . . . 85

4.2. Computational abstraction of the system . . . 85

4.3. CUPIC1D1V_PP implementation . . . 88

4.3.1. Initial conditions and steady state . . . 88

4.3.2. Particle weighting . . . 90

4.3.3. Poisson solver . . . 92

4.3.4. Particle mover . . . 95

4.3.5. Particle injection and boundary effects . . . 96

4.4. Comparison with fluid models . . . 99

4.5. Conclusion . . . 100

5. PIC simulation of a cylindrical Langmuir probe (CUPIC1D2V_CP) **101**

5.1. Introduction . . . 101

5.2. Computational abstraction of the system . . . 101

5.3. Differences between CUPIC1D2V_CP and CUPIC1D1V_PP . . . 104

5.3.1. Initial conditions . . . 104

5.3.2. Particle weighting . . . 104

5.3.3. Poisson solver . . . 105

5.3.4. Particle mover . . . 105

5.3.5. Particle injection . . . 107

5.4. Hybrid code optimisation . . . 109

5.5. Radial to Orbital motion transition . . . 110

5.6. Conclusion . . . 113

III Final remarks & Conclusions 115

6. Summary, contributions and future perspectives 117
6.1. Introduction . . . 117
6.2. Summary . . . 117
6.3. Contributions . . . 119
6.4. Future perspectives . . . 120
6.5. Conclusions . . . 121

IV Appendixes 123

A. Fluid approximation in plasmas: Boltzmann equation and its first moments. 125
A.1. Boltzmann transport equation . . . 125
A.2. First and second moments of the Boltzmann equation . . . 126

B. CUPIC1D1V_PP sources 129
B.1. Main module . . . 129
B.2. Initialisation module . . . 131
B.3. Mesh module . . . 143
B.4. Particles module . . . 147
B.5. Boundary conditions module . . . 149
B.6. Diagnostic . . . 154
B.7. CUDA module . . . 162
B.8. Extra headers . . . 163
B.9. Additional files . . . 164

C. CUPIC1D2V_CP sources 167
C.1. Main module . . . 167
C.2. Initialisation module . . . 169
C.3. Mesh module . . . 182
C.4. Particles module . . . 187
C.5. Boundary conditions module . . . 188
C.6. Diagnostic . . . 194
C.7. CUDA module . . . 201
C.8. Extra headers . . . 202
C.9. Additional files . . . 203

Bibliography 205

List of Figures

1.1. Detail of a small fraction of the Veil nebula 11
1.2. Classification of plasmas 12
1.3. Shielding of a charged ball introduced inside a plasma 13
1.4. Biased metallic grid being shielded by a plasma 14
1.5. Typical length of the Debye shielding 15
1.6. Structure of the contact of a plasma with an infinite planar metallic surface 16
1.7. Quasineutral solution for the potential (planar case) 19
1.8. Evolution of the ion flux along the presheath (planar case) 20
1.9. Sheath solution for Poisson's equation (planar case) 22
1.10. Complete solution for Poisson's equation (planar case) 23
1.11. Spherical Langmuir probe on board ESA's space vehicle Rosetta 24
1.12. Current voltage characteristic of a Langmuir probe 25

2.1. Cylindrical probe. Design and actual implementation in a laboratory plasma 30
2.2. Radial vs orbital probe theories 31
2.3. Mott-Smith and Langmuir model 32
2.4. TSL and OML approximations for the Mott-Smith and Langmuir theory 34
2.5. Classification of ion's orbits in the Bernstein and Rabinowitz model 38
2.6. Limits of integration in E, J^2 phase space for the Bernstein and Rabinowitz model 39
2.7. Geometrical presheath mechanism in cylindrical probes 43
2.8. Solutions of the ABR model for a cylindrical probe 45
2.9. Initial condition of the potential for the Fernández Palop's model ($r_p \lesssim \lambda_D$) 48
2.10. Solutions of Fernández Palop's model ($r_p \lesssim \lambda_D$) 48
2.11. Initial condition of the potential for the Fernández Palop's model ($r_p \gg \lambda_D$) 49
2.12. Solutions of Fernández Palop's model ($r_p \gg \lambda_D$) 50
2.13. Potential profile and $I - V$ characteristic with Morales Crespo's fitting coefficients 53
2.14. Sonin-plot for ABR and OML theories 55

3.1. Classification of codes used to simulate plasmas 59
3.2. General scheme of a particle simulation 60
3.3. Graphic interpretation of cyclic boundary conditions in 2D 64
3.4. Force evaluation in PIC codes 65

3.5. Particle weighting in 2D PIC simulations 66
3.6. Nearest-grid-point (NGP) weighting scheme 68
3.7. Cloud-in-cell (CIC) / Particle-in-cell (PIC) weighting scheme 69
3.8. Leap-frog integration scheme 71
3.9. Effusion of particles through a wall 72
3.10. Thermal vs drift velocity driven flux 73
3.11. CPU vs GPU approach to multicore processors 77
3.12. CUDA enabled automatic scalability 78
3.13. CUDA thread hierarchy 79
3.14. CUDA memory hierarchy 81
3.15. CUDA heterogeneous programming model 82

4.1. Computational abstraction of the simulation domain (planar probe) 86
4.2. Computational abstraction of the particle system (planar probe) 87
4.3. Dependence of the evolution of the simulation on the initial conditions 89
4.4. Scheme of the execution configuration of `particle_to_grid()` kernel. 90
4.5. Write collisions in `particle_to_grid()` kernel 91
4.6. Scheme of the execution configuration of `jacobi_iteration()` kernel. 93
4.7. Scheme of the `jacobi_iteration()` kernel. 94
4.8. Scheme of the `leap_frog_step()` kernel 95
4.9. Appearance of a source sheath during the simulation of a planar probe 97
4.10. Calibration of the ion flux in `CUPIC1D1V_PP` 98
4.11. Comparison between PIC simulation and fluid model (planar probe) 99

5.1. Computational abstraction of the simulation domain (cylindrical probe) 102
5.2. Computational abstraction of the particle system (cylindrical probe) 103
5.3. Calibration of the ion flux in `CUPIC1D2V_CP` 108
5.4. Comparison between the electron density by using particles or the fluid approximation . . 110
5.5. Dependence of the Sonin-plot on the dimensionless probe radius 111
5.6. Dependence of the ordinate of the Sonin-plot on the ion to electron temperature ratio . . 112
5.7. Dependence of the ordinate of the Sonin-plot on the ion to electron mass ratio 112

List of Tables

2.1. Morales Crespo's fitting coefficients for a cylindrical probe 52

3.1. Physical systems studied via particle simulations and computer experiments 62
3.2. Performance of intel processors along history 74
3.3. Run times for sequential particle simulations 75

List of Source Codes

3.1. Example of a simple CUDA kernel (VecAdd) . . . 78
3.2. Example of a simple CUDA kernel (MatAdd) . . . 79
3.3. Example of a simple CUDA kernel (MatAdd any size) . . . 80

B.1. CUPIC1D1V_PP source file main.cu . . . 129
B.2. CUPIC1D1V_PP source file init.cu . . . 131
B.3. CUPIC1D1V_PP source file init.h . . . 142
B.4. CUPIC1D1V_PP source file mesh.cu . . . 143
B.5. CUPIC1D1V_PP source file mesh.h . . . 147
B.6. CUPIC1D1V_PP source file particles.cu . . . 147
B.7. CUPIC1D1V_PP source file particles.h . . . 149
B.8. CUPIC1D1V_PP source file cc.cu . . . 149
B.9. CUPIC1D1V_PP source file cc.h . . . 154
B.10. CUPIC1D1V_PP source file diagnostic.cu . . . 154
B.11. CUPIC1D1V_PP source file diagnostic.h . . . 161
B.12. CUPIC1D1V_PP source file cuda.cu . . . 162
B.13. CUPIC1D1V_PP source file cuda.h . . . 163
B.14. CUPIC1D1V_PP source file stdh.h . . . 163
B.15. CUPIC1D1V_PP source file random.h . . . 164
B.16. CUPIC1D1V_PP source file dynamic_sh_mem.h . . . 164
B.17. CUPIC1D1V_PP compilation file makefile . . . 164
B.18. CUPIC1D1V_PP input file input_data . . . 165

C.1. CUPIC1D2V_CP source file main.cu . . . 167
C.2. CUPIC1D2V_CP source file init.cu . . . 169
C.3. CUPIC1D2V_CP source file init.h . . . 181
C.4. CUPIC1D2V_CP source file mesh.cu . . . 182
C.5. CUPIC1D2V_CP source file mesh.h . . . 186
C.6. CUPIC1D2V_CP source file particles.cu . . . 187
C.7. CUPIC1D2V_CP source file particles.h . . . 188
C.8. CUPIC1D2V_CP source file cc.cu . . . 188
C.9. CUPIC1D2V_CP source file cc.h . . . 193

C.10. CUPIC1D2V_CP source file diagnostic.cu 194
C.11. CUPIC1D2V_CP source file diagnostic.h 200
C.12. CUPIC1D2V_CP source file cuda.cu 201
C.13. CUPIC1D2V_CP source file cuda.h 202
C.14. CUPIC1D2V_CP source file stdh.h 202
C.15. CUPIC1D2V_CP source file random.h 203
C.16. CUPIC1D2V_CP source file dynamic_sh_mem.h 203
C.17. CUPIC1D2V_CP compilation file makefile 203
C.18. CUPIC1D2V_CP input file input_data 204

Agradecimientos/Acknowledgements

"Si he logrado ver más lejos, ha sido
porque he subido a hombros de gigantes"

Sir Isaac Newton

Como la gran mayoría de logros que conseguimos a lo largo de nuestra vida, la finalización de una tesis doctoral no sería posible sin la presencia, ayuda y apoyo de multitud de personas. Por lo tanto, es de justicia dedicar las primeras lineas de este trabajo a agradecer a todas las que lo han hecho posible.

En primer lugar me gustaría agradecer a José Ignacio, mi director de tesis, la confianza depositada en mi para llevar a cabo este trabajo. En todo este tiempo no solo me has apoyado y enseñado continuamente, aportando soluciones siempre que llegaba a un callejón sin salida, sino que también me has inculcado que hay cosas más importantes que el trabajo. Han sido incontables las horas que hemos compartido discutiendo detalles sobre la investigación, la fabricación de gafas de madera, o la calidad del café en la cafetería. Espero que en el futuro sean muchas más.

También me gustaría dar las gracias al resto de miembros del grupo de investigación. A Jerónimo, cuya experiencia parece ser infinita. Siempre has tenido un buen consejo que ofrecerme, y tus directrices han sido fundamentales para desarrollar con éxito una de las partes de este trabajo que más me apasiona, la docencia. A Juan Manuel, compañero de fatigas en los congresos, porque siempre que pasas por el laboratorio acabas levantándome el ánimo. A todos, porque espero tener la suerte de trabajar con vosotros muchos años.

Como no, también tengo que dar las gracias al resto de compañeros del departamento de física. A Rut, por las discusiones de gran talante científico que hemos mantenido. A los compañeros de "el café", porque sin ellos desconocería el significado de términos como "godovi" (espero haberlo escrito bien). Al resto de profesores, personal y doctorandos del departamento, porque da gusto estar rodeado de un grupo de personas como vosotros en el trabajo.

I also would like to thank the people I have met in Ljubljana during my stay there, for their warm welcome and because thanks to them I had a great time in Slovenia. To Prof. Tomaž Gyergyek because without him my stay there had not being possible. To Jernej and Boris for all the mountain tea shared while talking about anything. To the rest of the people from the Reaktor center for all the time spent together.

Y si hay mucha gente a la que agradecer en el plano profesional, hay mucha más en el plano personal. En primer lugar a mis padres, Antonio y Pilar, y mis hermanas, Miriam y Andrea, y al resto de mi familia, por apoyarme desde que tengo uso de razón, incluso cuando decidí estudiar esa carrera de locos, y luego un máster, y luego un doctorado... Sin todos ellos, simplemente, no sería lo que soy.

A mi segunda familia, mis amigos, porque tengo la grandísima suerte de contar con tantos que si los nombrase a todos necesitaría añadir otro apéndice a la tesis. No importa como de agobiado, cabreado o desanimado estuviese, vosotros siempre habéis estado ahí.

Estoy seguro de que dejo muchas personas y cosas que agradecer, pero no está bonito que los agradecimientos sean lo más extenso de una tesis. Así pues, gracias a todos, por haber sido gigantes a los que subirme.

Resumen

Actualmente, la física de plasmas constituye una parte importante de la investigación en física que está siendo desarrollada. Su campo de aplicación varía desde el estudio de plasmas interestelares y cósmicos, como las estrellas, las nebulosas, el medio intergaláctico, etc.; hasta aplicaciones más terrenales como la producción de microchips o los dispositivos de iluminación. Resulta particularmente interesante el estudio del contacto de una superficie metálica con un plasma. Siendo la razón que, la dinámica de la interfase formada entre un plasma imperturbado y una superficie metálica, resulta de gran importancia cuando se trata de estudiar problemas como: la implantación iónica en una oblea de silicio, el grabado por medio de plasmas, la carga de una aeronave cuando atraviesa la ionosfera y la diagnosis de plasmas mediante sondas de Langmuir.

El uso de las sondas de Langmuir está extendido a través de multitud de aplicaciones tecnológicas e industriales como método de diagnosis de plasmas. Algunas de estas aplicaciones han sido mencionadas justo en el párrafo anterior. Es más, su uso también es muy popular en la investigación en física de plasmas, por ser una de las pocas técnicas de diagnosis que proporciona información local sobre el plasma. El equipamiento donde es habitualmente implementado varía desde plasmas de laboratorio de baja temperatura hasta plasmas de fusión en dispositivos como tokamaks o stellerators. La geometría más popular de este tipo de sondas es cilíndrica, y la principal magnitud que se usa para diagnosticar el plasma es la corriente recogida por la sonda cuando se encuentra polarizada a un cierto potencial. Existe un interes especial en diagnosticar por medio de la medida de la corriente iónica recogida por la sonda, puesto que produce una perturbación muy pequeña del plasma en comparación con el uso de la corriente electrónica.

Dada esta popularidad, no es de extrañar que grandes esfuerzos se hayan realizado en la consecución de un modelo teórico que explique el comportamiento de una sonda de Langmuir inmersa en un plasma. Hay que remontarse a la primera mitad del siglo XX para encontrar las primeras teorías que permiten diagnosticar parámetros del plasma mediante la medida de la corriente iónica recogida por la sonda de Langmuir. Desde entonces, las mejoras en estos modelos y el desarrollo de otros nuevos ha sido una constante en la investigación en física de plasmas. No obstante, todavía no está claro como los iones se aproximan a la superficie de la sonda. Las dos principales, a la par que opuestas, aproximaciones al problema que están ampliamente aceptadas son: la radial y la orbital; siendo el problema que ambas predicen diferentes valores para la corriente iónica. Los experimentos han arrojado resultados de acuerdo con ambas teorías, la radial y la orbital; y lo que es más importante, una transición entre ambos ha sido recientemente observada.

La mayoría de los logros conseguidos a la hora de comprender como los iones caen desde el plasma hacia la superficie de la sonda, han sido llevados a cabo en el campo de la dinámica de fluidos o la teoría cinética. Por otra parte, este problema puede ser abordado mediante el uso de simulaciones de partículas. La principal ventaja de las simulaciones de partículas sobre los modelos de fluidos o cinéticos es que proporcionan mucha más información sobre los detalles microscópicos del movimiento de las partículas, además es relativamente fácil introducir interacciones complejas entre las partículas. No obstante, estas ventajas no se obtienen gratuitamente, ya que las simulaciones de partículas requieren grandísimos recursos. Por esta razón, es prácticamente obligatorio el uso de técnicas de procesamiento paralelo en este tipo de simulaciones.

El vacío en el conocimiento de las sondas de Langmuir, es el que motiva nuestro trabajo. Nuestra aproximación, y el principal objetivo de este trabajo, ha sido desarrollar una simulación de partículas que nos permita estudiar el problema de una sonda de Langmuir inmersa en un plasma y que está negativamente polarizada con respecto a éste. Dicha simulación nos permitiría estudiar el comportamiento

de los iones en los alrededores de una sonda cilíndrica de Langmuir, así como arrojar luz sobre la transición entre las teorías radiales y orbitales que ha sido observada experimentalmente.

Justo después de esta sección introductoria, el resto de la tesis está dividido en tres partes tal y como sigue:

- La primera parte está dedicada a establecer los fundamentos teóricos de las sondas de Langmuir. En primer lugar, se realiza una introducción general al problema y al uso de sondas de Langmuir como método de diagnosis de plasmas. A continuación, se incluye una extensiva revisión bibliográfica sobre las diferentes teorías que proporcionan la corriente iónica recogida por una sonda.

- La segunda parte está dedicada a explicar los detalles de las simulaciones de partículas que han sido desarrolladas a lo largo de nuestra investigación, así como los resultados obtenidos con las mismas. Esta parte incluye una introducción sobre la teoría que subyace el tipo de simulaciones de partículas y las técnicas de paralelización que han sido usadas en nuestros códigos. El resto de esta parte está dividido en dos capítulos, cada uno de los cuales se ocupa de una de las geometrías consideradas en nuestras simulaciones (plana y cilíndrica). En esta parte discutimos también los descubrimientos realizados relativos a la transición entre el comportamiento radial y orbital de los iones en los alrededores de una sonda cilíndrica de Langmuir.

- Finalmente, en la tercera parte de la tesis se presenta un resumen del trabajo realizado. En este resumen, se enumeran brevemente los resultados de nuestra investigación y se han incluido algunas conclusiones. Después de esto, se enumeran una serie de perspectivas futuras y extensiones para los códigos desarrollados.

Abstract

Nowadays, plasma physics constitutes an important part of the physics research that is currently being developed. Its field of applicability ranges from the study of interestelar and cosmic plasmas as stars, nebulae, intergalactic medium, etc.; to more down-to-earth applications as microchip manufacturing or lighting devices. It results of particular interest the study of the contact of a metallic surface with a plasma. The reason being that, the dynamics of the interphase formed between an unperturbed plasma and a metallic surface, results of great importance when it comes to study problems such as: ion implantation in a silicon wafer, plasma etching, charge of a spacecrafts when crossing the ionosphere and plasma diagnosis with Langmuir probes.

The use of Langmuir probes is widespread across lots of technological and industrial applications as a plasma diagnosing technique. Some of this applications have just been mentioned in the previous paragraph. Moreover, it is also very popular in plasma physics research, as it is one of the few diagnosing techniques that provides local information about the plasma. The equipment where it is commonly implemented varies from low temperature laboratory plasmas to fusion plasmas in devices like tokamaks or stellarators. The most popular geometry of such probes is cylindrical, and the main magnitude that it is used to diagnose a plasma is the current collected by the probe when biased at a certain voltage. There is a remarkable interest in diagnosing by measuring the ion current collected by the probe, since it produce very little perturbation of the plasma when compared to the use of the electron current.

Due to such popularity, it is not strange that great efforts have been made in the pursuit of a theoretical model that explains the behaviour of a Langmuir probe immersed in a plasma. We have to go back to the first half of the XX century to find the firsts theories that allow us to diagnose plasma parameters by measuring the ion current collected by a Langmuir probe. Since then, the improvements of these models and the development of new ones has been a constant in the plasma physics research. Nevertheless, it is still not clear how ions approach the surface of the probe. The two main, and opposite, frameworks that are widely accepted are: the radial and the orbital one; being the problem that they predict different values for the ion current. Experimentalists have found results in accordance to both, the radial and orbital theories; but more important, it has been recently found a transition between both of them.

Most of the achievements accomplished to figure out how ions fall from the plasma to the surface of a probe, have been developed in the field of the fluid dynamics or kinetic theory. On the other hand, this problem can be tackled by using particle simulations. The main advantages of particle simulations over fluid or kinetic models are that they provide much more information about the microscopic details of the movement of the particles, and that it is relatively easy to introduce complex interactions between particles. However, this advantages come at a price, and particle simulations are extremely resource demanding. Because of that reason, it is almost mandatory to use parallelisation techniques in this kind of simulations.

It is the void in the Langmuir probe knowledge, which motivates our work. Our approach here, and the main objective of this work, has been to develop a particle simulation that allows us to study the problem of a Langmuir probe immersed into a plasma and negatively biased with respect to it. This simulation would allow us to study the behaviour of ions in the surroundings of a cylindrical Langmuir probe, and to shed light in the experimentally found transition between the radial and orbital theories.

Right after this introductory section, the rest of the thesis is divided into three parts as follows:

- The first part is devoted to establish the theoretical foundations of Langmuir probes. First, a general introduction to the problem and the use of Langmuir probes as a plasma diagnosing technique is made. Then, an extensive bibliographic review about the different theories that provides the ion

current collected by a cylindrical probe is included.

- The second part is devoted to explain the details of the particle simulations developed along our research as well as the results obtained with them. This part includes an introduction about the theory behind particle simulations and the parallelisation techniques that have been used in our codes. The rest of this part is divided into two chapters, each one concerning one of the geometries considered in our simulations (planar and cylindrical). In this part we discuss our findings in the transition between the radial and orbital behaviour of ions in the vicinity of a cylindrical Langmuir probe.
- Finally, in the third part of the thesis a summary of the work is presented. In this summary the results of the research are briefly enumerated and some conclusions are included. After that, future research outlooks and extensions for the developed codes are outlined.

"But one day Langmuir came in triumphantly and said he had it. He pointed out that the 'equilibrium' part of the discharge acted as a sort of sub-stratum carrying particles of special kinds, like high-velocity electrons from thermionic filaments, molecules and ions of gas impurities. This reminds him of of the way blood plasma carries around red and white corpuscles and germs. So he proposed to call our 'uniform discharge' a 'plasma'. Of course we all agreed."

Harold M. Mott-Smith – History of "Plasmas"

Part I

Theoretical Foundations

Chapter 1

Plasmas, sheaths, probes and diagnosis

1.1. Introduction

The first studies about discharges in rarified gases, where performed in the General Electric Research Lab in Schenectady, New York in the 1920s. These studies led to the development of a new branch of physics that is nowadays known as plasma physics. Actually, it was there where the term “plasma” was coined by Irving Langmuir [1].

In this chapter we are going to introduce some of the basic concepts on which our research rely. Starting with the definition of what a plasma is, and some of the peculiarities of this manifestation of matter. Then we deal with the question of what happens when we introduce an object into a plasma. Our interest will be focused in the use of metallic objects, as this leads to the Langmuir probe concept. Finally we will discuss how this probes can be used in order to diagnose different plasma parameters.

1.2. What is a Plasma?

The universe is composed of a mixture of energy and matter. It is well known that more than 99% of the matter of the observable universe (that is, excluding dark matter) exists in the state known as plasma. But, what is exactly the plasma state of the matter? Well, the definition is rather simple, a **plasma** is a gas which: is totally or partially ionised, macroscopically quasineutral and exhibits a collective behaviour. Avoiding for now the quasineutral and collective behaviour conditions, the rather simple definition of plasmas given before, leads us to the conclusion that the stars are almost 100% plasma. Now it easy to see why the vast majority of the observable universe is composed of plasma. But things does not finish there: interplanetary matter is almost 100% ionised hydrogen, interstellar matter it is also composed of huge percentages of ionised hydrogen and helium and nebulae are colossal clouds of mostly ionised gases resulting from the explosion of a dying star (see Fig. 1.1).

Figure 1.1: Detail of a small fraction of the Veil nebula, approximately two light-years across. The size of the whole nebula is around 110 light-years across. Image Credit: NASA/ESA/Hubble Heritage Team.

Even though all the examples given above are extremely far away from us, in our small human length scale, we do not have to go to outer space

in order to find a plasma. The ionosphere, which is the part of the earth's atmosphere that protects us from the solar winds, is mainly composed of plasma. By the way, the solar winds are also mainly plasma. But we can find natural plasmas even closer. During a storm, clouds charge themselves because of the friction with the air. Due to this charged clouds, an electric field arises between the ground and the clouds. When this electric field reaches what is called the dielectric breakdown value, the air between the clouds and the ground ionises, allowing an extremely fast discharge of the cloud. What has been just described is a lightning, which is also a plasma. But we can take one last step closer to our experience. Flames in a fire are a plasma as well. The air in the surrounding of a fire is heated so much that some electrons are pulled off the air molecules, resulting again in an ionised gas.

Obviously, with such a diversity of plasmas found in the nature we can safely say that, not all plasmas are the same. Even more if we take account of all the different man-produced plasmas, from low consumption bulbs to thermonuclear fusion plasmas produced in tokamaks and stellarators. In order to create a plasma, we have to communicate enough energy to a gas in order to ionise its molecules. This energy could come from: nuclear fusion reactions (like in the sun), collisions with energetic particles (like in the ionosphere), heating by chemical reactions (like in a flame), electric discharges (like in a lightning), etc.

Regardless of the source of energy that produce them, basically all the plasmas are the same, a gas composed of neutral atoms or molecules, ions and free electrons. That is the reason why plasmas are usually classified by their density and temperature of charged particles. In particular, the electron density and temperature are usually used in that classification. Some of the examples that have been previously used can be seen classified in the graph of Fig. 1.2.

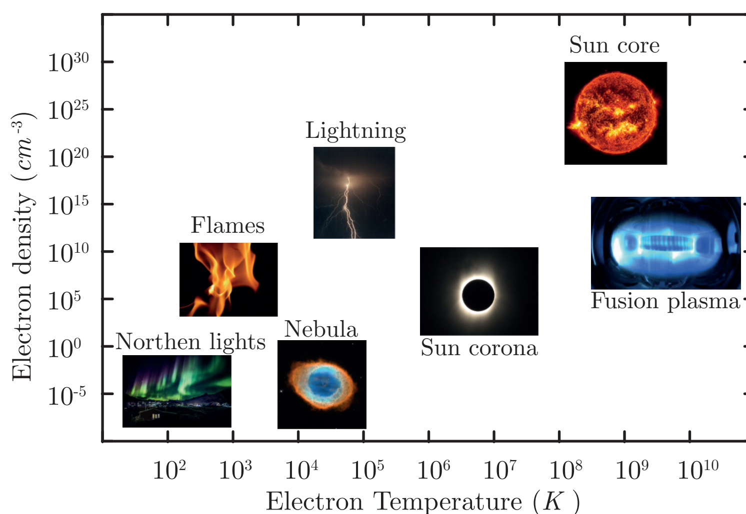


Figure 1.2: Classification of plasmas in terms of their electron number density and temperature.

Precisely because of the presence of charged particles, the plasma dynamic is highly conditioned by Maxwell's equations of electromagnetism. This fact is directly related to the quasineutral and collective behaviour conditions that we mentioned in the plasma definition. First, when at equilibrium, a plasma has to be electrically neutral. That is, if we only consider singly ionised ions, $n_e \approx n_i$, where n_e and n_i are the electron and ion number density respectively. An accidental local accumulation of charge would cause an electric field to arise, causing the more populated species to be repelled by themselves. This process would continue until the electric equilibrium is reached and the quasineutral condition holds. The second condition that appears into the plasma definition, *i.e.* collective behaviour, is related to how charged particle motion inside a plasma is not exclusively due to local conditions, but to the state of the plasma at remote regions.

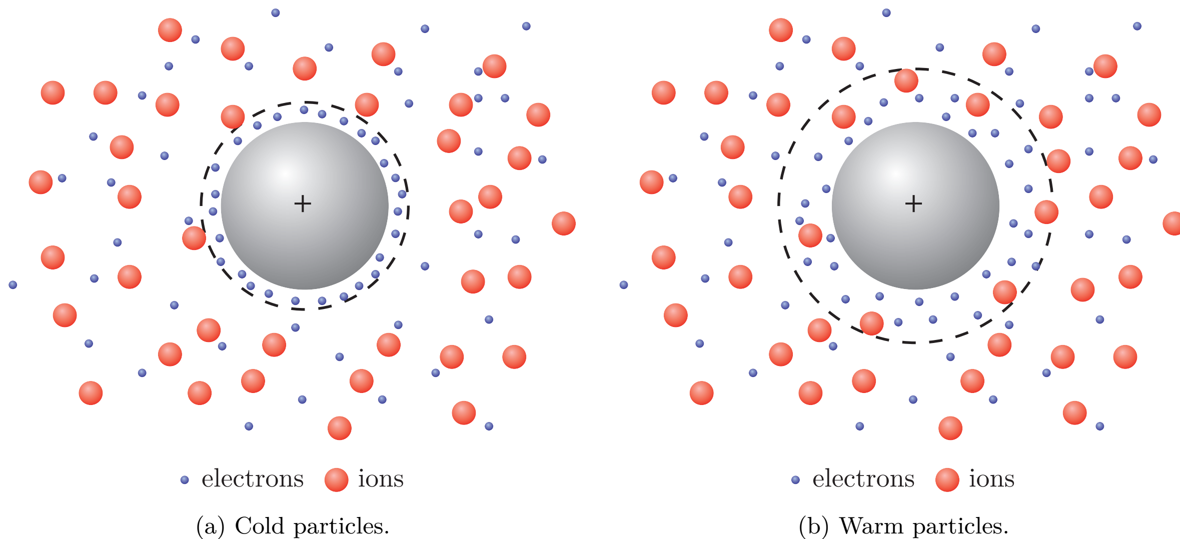
It is important to note that, the quasineutral conditions that has been just introduced ($n_e \approx n_i$), only holds in an unperturbed plasma. We will discuss this fact deeply in the next section where we will introduce the sheath concept. Another remark that needs to be made is that, in most cases, the effects of the magnetic fields created by the particles that constitute the plasma are negligible. When this happens we do not have to take into account the four Maxwell's equations, instead, we only consider Gauss Law to tackle electrostatic effects. In this case we have what is called an electrostatic plasma. From now on we will only consider electrostatic plasmas, contrary to magnetised ones, where an external magnetic field is imposed into the plasma and the whole set of Maxwell's equations have to be taken into consideration.

1.3. Debye shielding

One of the main abilities of a plasma has just been outlined in the previous section while talking about the quasineutral condition. We refer to the ability of shielding electric fields. This ability is not surprising at all. A plasma is composed of free charged particles, so it conducts electricity, and as any other good conductor of electricity, it tries to avoid electric fields in its inside. Let us think this more carefully.

For the sake of simplicity we are going to consider a plasma consisting only of electrons and singly ionised ions, and assume that both are completely cold. In this situation we can think, what would happen if we introduce a charged ball, *e.g.* positively charged, inside the plasma?. First, the ball would create a perturbation of the electric potential inside the plasma, developing an electric field. This electric field would attract electrons towards the ball and repel ions. This process would go on until an extremely thin layer of electrons would form around the ball in order to fully shield it. The charge of this thin layer of electrons would be exactly the same as the one of the ball, but obviously with oposite sing. Also, as particles have no thermal motion, the electrons of the layer would remain "sticked" to the ball. Finally, the potential perturbation that the ball introduced into the plasma would have been completely shielded. This can be seen with the help of the Gauss law. The net charge inside the imaginary surface that is just outside the shielding layer of electrons (see Fig. 1.3a) is exactly zero. So, the electric field created by the ball and the layer of electrons outside this surface is zero, and the ball becomes "invisible" to the rest of the plasma.

The previously made assumption of completely cold particles is not very realistic, especially when dealing with electrons. The tiny mass of electrons, provides them an incredibly high mobility, so they always have thermal motion until some extent. Nevertheless, the picture is almost the same that the one described in the previous paragraph. The difference is that, when the shielding layer is being formed, the thermal motion of electrons provides energy to try to scape from the potential well created by the ball. In fact, electrons which have a kinetic energy larger than the potential energy they have by the side of the ball, could escape from its attraction and go back into the unperturbed plasma. So, the net effect of considering warm particles is that a finite length is needed in order to shield the ball (see Fig. 1.3b). Also, as some of the electrons shielding the ball are going to be able to escape from its attraction, the shielding is no longer going to be perfect, and a small amount of the perturbation is going to leak into the plasma.



(a) Cold particles.

(b) Warm particles.

Figure 1.3: Shielding of a charged ball introduced inside a plasma.

The effect that we have been describing in this section is called **Debye shielding**, and can be defined as the ability of a plasma to shield, more or less effectively, electrical perturbations. Let us do some math in order to study this property in a more quantitative fashion. In particular, we are interested in quantifying how well the plasma can shield a perturbation of the electric potential, and how much of the perturbation is leaked into the plasma. As we have seen, when particles are completely cold, the shielding would be perfect. For that reason we are going to consider the case of warm particles. Let us

also consider a slightly more realistic situation.

Suppose we have the same plasma that we described previously, composed of electrons and singly ionised ions, and now we are going to introduce in it an infinite planar metallic grid biased with respect to the plasma. The grid is 100% transparent, so particles can cross it without interacting with it. This situation can be seen schematically in Fig. 1.4. We are looking for how the perturbation of the electric potential is shielded until the electric field in the bulk plasma is negligible. In order to accomplish that, we are going to solve Poisson's equation, $\nabla^2\varphi = -\rho/\varepsilon_0$. Obviously, the only relevant dimension in our problem is the one perpendicular to the plane of the grid. So we can write Poisson's equation as:

$$\frac{d^2\varphi(x)}{dx^2} = -\frac{e}{\varepsilon_0}(n_i(x) - n_e(x)) \tag{1.1}$$

where $\varphi(x)$ is the electric potential, x the position with respect to the plane of the grid, e the elementary charge, $n_i(x)$ the number density of ions, $n_e(x)$ the number density of electrons and ε_0 the permittivity of free space.

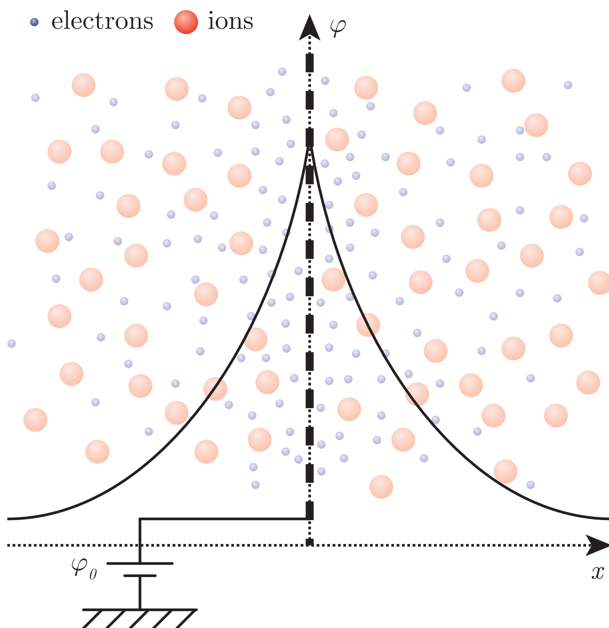


Figure 1.4: Biased metallic grid being shielded by a plasma composed of singly ionised ions and warm electrons.

The first thing we need, in order to solve Eq. (1.1), is the value of the particle densities $n_i(x)$ and $n_e(x)$. As we are going to consider the much more meaningful case of warm particles, we can assume that the distribution function of electrons in the presence of an electric potential $\varphi(x)$ is the one corresponding to the thermal equilibrium:

$$f_e(x, v) = A\ \exp\left(-\frac{m_e v^2/2 - e\varphi(x)}{k_B T_e}\right) \tag{1.2}$$

m_e being the mass of electrons, T_e their temperature, k_B the Boltzmann constant, v the velocity with which electrons move and A a normalisation constant. We are not going to demonstrate here the expression in Eq. (1.2). Nevertheless its meaning is rather obvious from an intuitive point of view. There are fewer particles where the potential energy is higher, as there are fewer particles with enough kinetic energy in order to reach that point. Now, by integrating Eq. (1.2) for every possible velocity, the electron density can be obtained as:

$$n_e(x) = n_{e0}\ \exp\left(\frac{e\varphi(x)}{k_B T_e}\right) \tag{1.3}$$

On the other hand, during the time needed by electrons to shield the perturbation, the movement of ions is going to be negligible. Due to the larger mass of ions, $m_i \gg m_e$, they have a mobility much smaller than electrons. For that reason, we can assume that the ion density is not going to change noticeably from its value at the unperturbed plasma, $n_i(x) \approx n_i(x \to \infty) = n_{i0}$. Also, at the unperturbed plasma the quasineutral condition holds, so the electron and ion densities there must be equal, $n_{e0} = n_{i0}$. Now, if we introduce this conditions along with Eq. (1.3) in Eq. (1.1) we get:

$$\frac{d^2\varphi(x)}{dx^2} = -\frac{e n_{e0}}{\varepsilon_0}\left[1 -\ \exp\left(\frac{e\varphi(x)}{k_B T_e}\right)\right] \tag{1.4}$$

We are not interested here in the details of how the potential drops near the grid. Instead, we would like to know how the potential varies at large values of x and how much of it leaks into the plasma. It is clear that at some point, as we increase x, the potential will reach a value that fulfils the condition $e\varphi(x)/(k_B T_e) \ll 1$. With that in mind, if we expand the exponential term of Eq. (1.4) in a Taylor series:

$$\exp\left(\frac{e\varphi(x)}{k_B T_e}\right) = \left[1 + \frac{e\varphi(x)}{k_B T_e} + \cdots\right] \approx 1 + \frac{e\varphi(x)}{k_B T_e} \tag{1.5}$$

and introducing Eq. (1.5) into Eq. (1.4) we obtain:

$$\frac{d^2\varphi(x)}{dx^2} = \frac{e^2 n_{e0}}{\varepsilon_0 k_B T_e}\varphi(x) \tag{1.6}$$

Now, the solution of Eq. (1.6) is easy to find, as long as we set the boundary conditions $\varphi(0) = \varphi_0$ and $\varphi(x \to \infty) = 0$. The solution can be written as:

$$\varphi(x) = \varphi_0 \exp\left(-\frac{|x|}{\lambda_D}\right) \tag{1.7}$$

where we have defined the quantity:

$$\boxed{\lambda_D = \sqrt{\frac{\varepsilon_0 k_B T_e}{e^2 n_{e0}}}} \tag{1.8}$$

This magnitude is called **Debye length**, and it is a measure of the length of the shielding of the perturbation. As can be seen in Fig. 1.5, the potential is almost shielded when we move away from the grid a distance of a few times λ_D. The larger this length the more the perturbation leaks into the plasma. As we have already stated, the ability to shield electric perturbations is one of the defining characteristics of a plasma. It is not casual that the parameters that we chose in order to classify plasmas in Fig. 1.2, appear in the definition of the Debye length. This parameters are the electron temperature and number density. As we can see in Eq. (1.8), the Debye length increases as the electron temperature is increased and decreases as the electron density is increased.

If we remember the situation shown in Fig. 1.3 it is easy to understand the dependence of the Debye length on those parameters. As we explained there, the broadening of the shielding layer of electrons was caused by the thermal energy of electrons. Even more, this energy allows some of them to escape completely from the potential well created by the ball. On the other hand, when the density is increased, electrons are more tightly packaged around the ball. So, less space is needed in order to fit enough electrons to shield the ball.

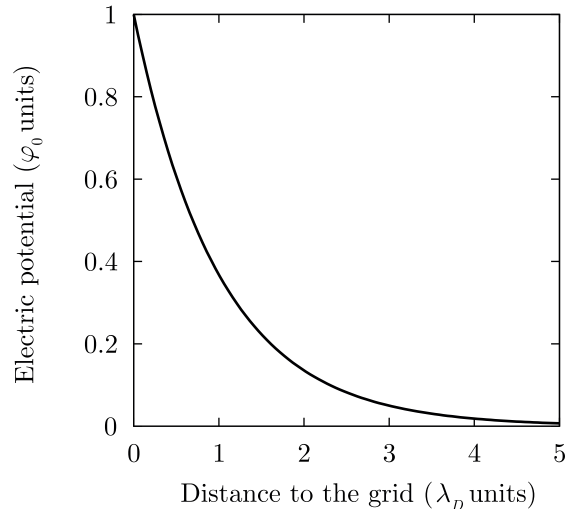


Figure 1.5: Shielding of the electric potential given by Eq. (1.7). It became negligible after a few λ_D.

The space occupied by the cloud of charge that shields the perturbation is called the **sheath**. Inside this zone positive and negative charge are not balanced in order to shield the perturbation, meaning that the quasineutral condition is not hold. As the conditions required for an ionised gas to be a plasma are no longer met, the sheath can not be properly called plasma. The structure of this zone will be further discussed in the following section, but we already know that its extension is several Debye lengths.

Speaking of the conditions for an ionised gas to be a plasma, we have to remember the "collective behaviour" condition introduced in the previous section. Let us further explain what this condition implies.

- First, we have to notice that the characteristic length of the plasma, let say L, has to be larger than the Debye length. That is, $\lambda_D \ll L$, in order for the plasma to be able to shield the perturbations introduced by the boundaries that confine it. Otherwise, quasineutrality could not be reached in the bulk plasma.
- On the other hand, we have to notice that in order to use statistical arguments, like the use of distribution functions like Eq. (1.2) and Eq. (1.3) or the Debye shielding concept itself, we need to

have a reasonably high number of particles. For that reason, the number N_D of particles inside a Debye sphere (sphere with radius equal to λ_D) has to be large enough. That number of particles is easily evaluated as $N_D = n_{e0}(4/3)\pi\lambda_D^3$. In case this number is not large enough, the gas is so diluted it can not exhibits a collective behaviour.

- And last, but not least, the characteristic time, τ, of the fastest process involved in the dynamics of the system, has to be smaller than the characteristic time that the plasma needs to shield any perturbation. For example, the larger collision frequency has to be smaller than what is called the **electron plasma frequency**, ω_{pe}. Otherwise, the dynamic of the system would be ruled mainly by other causes instead of electromagnetism, and the motion of particles would depends on local conditions instead of the collective state of the system.

  This electron plasma frequency can be defined as the inverse of the characteristic time that the Debye shielding needs to be developed when a perturbation is introduced into the plasma. Knowing that the characteristic length of the Debye shielding is λ_D, and the characteristic velocity of electrons due to its thermal motion is $v_{th} = \sqrt{k_B T_e/m_e}$, we have that $\omega_{pe} = v_{th}/\lambda_D = \sqrt{n_{e0}e^2/m_e\varepsilon_0}$.

Finally, we can say that a plasma is an ionised gas that fulfils the following conditions in order to show a collective behaviour:

$$\lambda_D \ll L \tag{1.9a}$$

$$N_D = n_{e0}\frac{4}{3}\pi\lambda_D^3 \gg 1 \tag{1.9b}$$

$$\omega_{pe} = \sqrt{\frac{n_{e0}e^2}{m_e\varepsilon_0}} \gg \frac{1}{\tau} \tag{1.9c}$$

1.4. Contact of a plasma and a metallic surface

Let us now study what happens when a metallic surface, *i.e.* a conductor, is introduced within a plasma. This will allow us to deepen the knowledge about the sheath zone, that we have just introduced, and its structure.This will also leads us to the definition of an electrostatic Langmuir probe.

We are going to consider the same plasma as before, consisting only of electrons and singly ionised ions. Into this plasma, we are going to introduce an infinite planar metallic surface, which will be considered to be perfectly absorbing. This means that, when any particle hits the surface, it is absorbed and it can not bounce off and go back into the plasma. We are also considering that the surface is negatively biased with a potential $\varphi_p < 0$ with respect to the plasma, as this are the biasing conditions our research is going to be focused on. Under this conditions, the surface is going to repel electrons and attract ions. Therefore, a charged cloud of particles, mostly consisting of ions in this case, is going to develop in front of the surface. This positively charged cloud is going to shield out the perturbation of the potential introduced by the surface. The main difference with respect to the previously used examples is that the particles that reach the surface are absorbed by it. Consequently, the conductor is going to drain a certain current density from the plasma. Because of the biasing potential we have chosen to use, this current is going to be mostly due to ions, which is the case we were looking for.

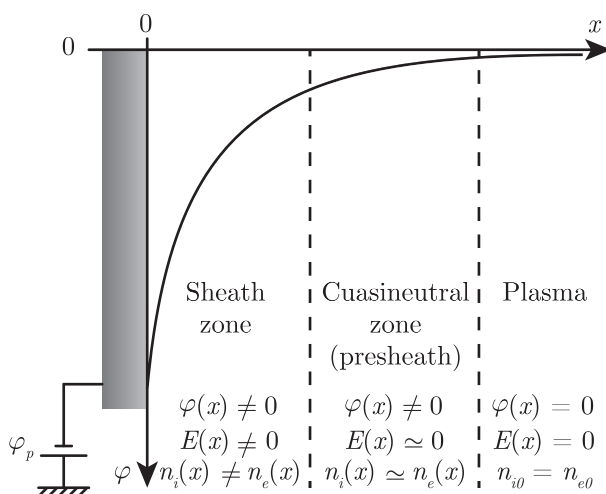


Figure 1.6: Structure of the contact between a plasma and an infinite planar metallic surface which is negatively biased with respect to the plasma. (biasing potential $\varphi_p < 0$)

In Fig. 1.6 a scheme of the structure of the potential is shown. As can be seen, a new zone has been introduced between the previously defined

sheath and the unperturbed plasma. This new zone is usually referred to as the preseath or the quasineutral zone. The **presheath** is necessary in order to connect the sheath and the plasma. It starts by the end of the sheath, once the quasineutral condition is hold, *i.e.* $n_i(x) \approx n_e(x)$. Precisely because of the quasineutral character of the presheath, the electric field in its inside is negligible. Nevertheless, the electric potential in this zone has not already reached the plasma potential, which we are using as the reference.

Now we are going to find the distribution of electric potential across the different zones defined in Fig. 1.6, in order to characterise them. The starting point is the same as the one in the previous section: Poisson's equation. Also, because of the same reasons that were given there, the only relevant dimension in our problem is the one perpendicular to the metallic surface.

$$\frac{d^2\varphi(x)}{dx^2} = -\frac{e}{\varepsilon_0}(n_+(x) - n_e(x)) \tag{1.10}$$

So, Eq. (1.10) is the one we have to solve. In order to do that, we need again the particles number densities.

Let us start with electrons. As the biasing potential chosen for the probe is negative, the potential profile developed between the surface and the plasma is retarding for electrons. That means that electrons are reflected by the surface. This fact allows us to consider electrons to be in thermal equilibrium with the electric field. So their distribution function can be written as:

$$f_e(x,\vec{v}) = n_{e0}\left(\frac{m_e}{2\pi k_B T_e}\right)^{3/2} \exp\left(-\frac{m_e(v_x+v_y+v_z)^2/2 - e\varphi(x)}{k_B T_e}\right) \tag{1.11}$$

By integrating Eq. (1.11) for every posible $\vec{v}$ values, we can obtain the electron density profile. But first, we have to notice something. Even though electrons are repelled by the conductor, it is posible that some of them have enough kinetic energy in the x dimension to overcome the potential well shown in Fig. 1.6. That is, electrons fulfilling the condition $m_e v_x^2/2 > -e\varphi_p$ at the plasma ($x \to \infty$), and moving towards the surface, are going to reach it. These electrons are going to be absorbed by the conductor; thus, they are not going back to the plasma. For that reason, at any point $x > 0$, we can find electrons with velocities $v_x < -\sqrt{2e(\varphi(x)-\varphi_p)/m_e}$ but not the corresponding $v_x > \sqrt{2e(\varphi(x)-\varphi_p)/m_e}$. The reason being that, electrons that would have such a positive velocity, were absorbed by the surface when they were coming from the plasma. So, the posible v_x values at any point $x > 0$ are: $-\infty < v_x < \sqrt{2e(\varphi(x)-\varphi_p)/m_e}$. While for the remaining components of the velocity we have: $-\infty < v_y < \infty$, $-\infty < v_z < \infty$. Finally, we can evaluate the electron number density as:

$$n_e(x) = \int_{-\infty}^{\sqrt{2e(\varphi(x)-\varphi_p)/m_e}} dv_x \int_{-\infty}^{\infty} dv_y \int_{-\infty}^{\infty} dv_z\ f(x,\vec{v}) \tag{1.12}$$

The integral in Eq. (1.12) can be easily solved and yields:

$$n_e(x) = \frac{n_{e0}}{2}\exp\left(\frac{e\varphi(x)}{k_B T_e}\right)\left(1 + \ \mathrm{erf}\left(\sqrt{e(\varphi(x)-\varphi_p)/k_B T_e}\right)\right) \tag{1.13}$$

where $\mathrm{erf}\,(x)$ is the error function, which is defined as follows:

$$\mathrm{erf}\,(x) = \frac{2}{\sqrt{\pi}}\int_0^x \exp\left(-s^2\right) ds \tag{1.14}$$

On the other hand, the surface is attracting and absorbing ions because of its negative biasing potential. Due to the continuous drain of ions from the plasma by the surface, their distribution function is so perturbed that we can no longer consider an expression like the one in Eq. (1.11). This means that ions are not in thermal equilibrium with the electric field, so we are going to use a fluid approximation in their description. The most simple balance moment equation that can be considered for the ion fluid can be written as [2]:

$$n_i(x)v_i(x)\frac{dv_i(x)}{dx} + \frac{e}{m_i}n_i(x)\frac{d\varphi(x)}{dx} = 0 \tag{1.15}$$

$v_i(x)$ being the ion fluid flow velocity and m_i is the ion mass. In this equation we state that ions move under the sole influence of the electrostatic force, and that we are in the low ionisation limit (see Appendix A for discussion). This are both reasonable assumptions in low pressure and temperature plasmas. Collisions are extremely rare in low pressure conditions, so their contribution to the dynamics of the motion of ions is insignificant. Also, because of the large mass and low temperature of ions, T_i, their thermal velocity $v_{i\text{th}} = \sqrt{2k_B T_i/m_i}$ is negligible compared to their flow velocity $v_i(x)$.

Eq. (1.15) can be integrated once, and knowing that $\varphi(x \to \infty) = 0$ and $v_i(x \to \infty) = 0$ we obtain the following energy conservation equation:

$$\frac{1}{2} m_i v_i^2(x) + e\varphi(x) = 0 \tag{1.16}$$

Besides, continuity equation for ions can be written as:

$$\frac{dn_i(x)v_i(x)}{dx} = -Zn_e(x) \tag{1.17}$$

where the right hand side of the equation takes into account the creation of ions because of the ionisation of neutrals by collisions with electrons, Z being the frequency of ionisation. It has to be noticed that, the sign of the right hand side of Eq. (1.17), is introduced to take into account the negative character of the flow velocity $v_i(x)$ (see Fig. 1.6). The ion flux, *i.e.* number of ions that cross a surface per unit time and area, can be written as:

$$j_i(x) = n_i(x)v_i(x) \tag{1.18}$$

Now Eq. (1.16) and Eq. (1.18) can be rearranged in order to write the ion density as:

$$n_i(x) = \frac{j_i(x)}{v_i(x)} = -j_i(x)\sqrt{-\frac{m_i}{2e\varphi(x)}} \tag{1.19}$$

Finally, we have expressions for the electron and ion density given by Eq. (1.13) and Eq. (1.19) respectively, so, Poissons' equation and continuity equation can be expressed as:

$$\frac{d^2\varphi(x)}{dx^2} = \frac{e}{\varepsilon_0}\left[j_i(x)\sqrt{-\frac{m_i}{2e\varphi(x)}} + \frac{n_{e0}}{2}\exp\left(\frac{e\varphi(x)}{k_B T_e}\right)\left(1 + \text{ erf}\left(\sqrt{e(\varphi(x) - \varphi_p)/k_B T_e}\right)\right)\right] \tag{1.20a}$$

$$\frac{dn_i(x)v_i(x)}{dx} = Z\frac{n_{e0}}{2}\exp\left(\frac{e\varphi(x)}{k_B T_e}\right)\left(1 + \text{ erf}\left(\sqrt{e(\varphi(x) - \varphi_p)/k_B T_e}\right)\right) \tag{1.20b}$$

For the sake of simplicity, we are going to transform Eqs. (1.20) in order to work in a dimensionless fashion. Accordingly, we introduce the following dimensionless variables:

$$X = \frac{x}{\lambda_D}; \quad \psi(X) = \frac{e\varphi(x)}{k_B T_e}; \quad N_i(X) = \frac{n_i(x)}{n_{e0}}; \quad V_i(X) = \frac{v_i(x)}{\lambda_D \omega_{pe}}; \quad \delta = \frac{Z}{\omega_{pe}}; \quad \gamma = \frac{m_i}{m_e} \tag{1.21}$$

So, inserting definitions (1.21) into Eqs. (1.20) we obtain the following dimensionless equations that describe the behaviour of the potential across the sheath and the quasineutral zone:

$$\frac{d^2\psi(X)}{dX^2} = \frac{1}{2}e^{\psi(X)}\left(1 + \text{ erf}\left(\sqrt{\psi(X) - \psi_p}\right)\right) + J_i(X)\sqrt{\frac{\gamma}{-2\psi(X)}} \tag{1.22a}$$

$$\frac{dJ_i(X)}{dX} = \frac{\delta}{2}e^{\psi(X)}\left(1 + \text{ erf}\left(\sqrt{\psi(X) - \psi_p}\right)\right) \tag{1.22b}$$

We are going to consider the case of low ionisation, which means $\delta \to 0$. In this case we are going to see how two different solutions of Eqs. (1.22) can be found, one for the quasineutral zone and the other for the sheath zone. Which solution it is found depends on the length scale chosen to solve the problem.

It has to be noticed that Eqs. (1.22) are an autonomous system of differential equations, *i.e.* they do not explicitly depend on the independent variable. In this case it can be proved that, if we find a solution $f(X)$ of the system, then $f(X + X_0)$ is also a solution, where $f(X)$ represents the electric potential, field, or the ion flux. This fact will allow us to arbitrarily choose the origin of positions, as we can perform any translation in the solutions that we find.

1.4.1. Quasineutral solution

To obtain the solution corresponding to the quasineutral zone, we are going to define a new length scale. This new length scale is not going to be characterised by the Debye length anymore, but by the ionisation mean free path. So, instead of using the dimensionless variable X we are going to use the variable $Y = \sqrt{\gamma}\delta X$. Obviously, Y is also dimensionless, and the introduction of its definition into Eqs. (1.22) yields:

$$\gamma\delta^2 \frac{d^2\psi(Y)}{dY^2} = \frac{1}{2}e^{\psi(Y)}\left(1 + \operatorname{erf}\left(\sqrt{\psi(Y) - \psi_p}\right)\right) + J_i(Y)\sqrt{\frac{\gamma}{-2\psi(Y)}} \tag{1.23a}$$

$$\frac{dJ_i(Y)}{dY} = \frac{1}{2\sqrt{\gamma}}e^{\psi(Y)}\left(1 + \operatorname{erf}\left(\sqrt{\psi(Y) - \psi_p}\right)\right) \tag{1.23b}$$

Now, if we take the limit $\delta \to 0$, the left hand side of Eq. (1.23a) becomes zero. This means that both terms on the right hand side of Eq. (1.22a) have to be numerically equal. If we go back to Eq. (1.10) it can be seen that these two terms corresponds to the electron and ion number density. Knowing that, it is clear why the presheath is also called the quasineutral zone, as the quasineutral condition is hold. Also, as we will see, most of the potential drop takes place in the sheath zone (see Fig. 1.6). This means that, inside the presheath, the potential is still far from the value at the metallic surface. So, in the presheath $\sqrt{\psi(Y) - \psi_p} \gg 1$ and so $\operatorname{erf}\left(\sqrt{\psi(Y) - \psi_p}\right) \approx 1$. Taking into account this two considerations, Eqs. (1.23) become:

$$J_i(Y) = -\sqrt{\frac{-2\psi(Y)}{\gamma}}e^{\psi(Y)} \tag{1.24a}$$

$$\frac{dJ_i(Y)}{dY} = \frac{e^{\psi(Y)}}{\sqrt{\gamma}} \tag{1.24b}$$

Finally, Eq. (1.24a) can be introduced into Eq. (1.24b) to obtain a single ordinary differential equation (ODE) for the potential:

$$\left(\frac{1}{\sqrt{-2\psi(Y)}} - \sqrt{-2\psi(Y)}\right)\frac{d\psi(Y)}{dY} = 1 \tag{1.25}$$

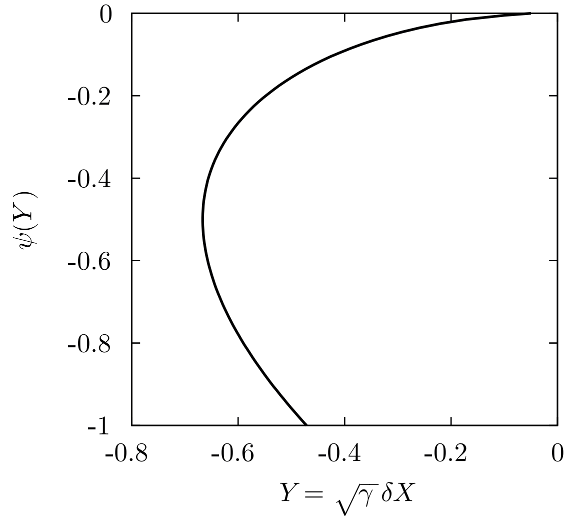


Figure 1.7: Quasineutral solution for the potential in the limit $\delta \to 0$, as given by Eq. (1.26).

Eq. (1.25) can be easily integrated as a Cauchy problem once, we have set the proper initial conditions. If we consider that the plasma is located at $Y = 0$ and the metallic surface at some point $Y < 0$, then the initial condition for the potential is $\psi(0) = 0$. Using this initial condition, the following implicit solution for the potential is obtained by direct integration:

$$Y = \frac{(-2\psi(Y))^{3/2}}{3} - \sqrt{-2\psi(Y)} \tag{1.26}$$

The quasineutral solution given by Eq. (1.26) is shown in Fig. 1.7. It can be seen that the potential departs from the value it has at the plasma, $\psi(0) = 0$, and starts decreasing approaching the biasing potential of the metallic surface, $\psi_p < 0$. However, at some point Y, after a small potential drop, the gradient of the potential goes to infinity. This means that the electric field diverges at some point before the potential has reached the value ψ_p. The point where the electric field diverges marks the end of the quasineutral zone, since the quasineutral condition does not hold. From this point, the quasineutral solution given by Eq. (1.26) is no longer valid and the sheath solution has to be used. In order to characterise the end point of the quasineutral zone we have to analyse Eq. (1.24b):

$$\frac{dJ_i(Y)}{dY} = \underbrace{\frac{dJ_i(\psi)}{d\psi}}_{\to 0}\underbrace{\frac{d\psi(Y)}{dY}}_{\to\infty} = -\frac{e^{\psi(Y)}}{\sqrt{\gamma}} \tag{1.27}$$

So, differentiating Eq. (1.24a) and equating to zero:

$$\frac{dJ_i(\psi)}{d\psi} = e^{\psi}\left(\frac{1}{\sqrt{-2\gamma\psi}} - \sqrt{\frac{-2\psi}{\gamma}}\right) = 0 \Rightarrow \psi = \psi_B = -\frac{1}{2} \tag{1.28}$$

We have found that the presheath ends when the potential reaches a value $\psi_B = -0.5$, as can be seen in Fig. 1.7. As we will see, this is also the starting point of the sheath, which solution can not be found if the initial condition for the potential is larger than ψ_B. The role of the preseath is to match the sheath solution with the plasma. It provides a zone where the potential can decrease from zero to $\psi_B = -0.5$ without violating the quasineutral condition. If we introduce the potential ψ_B into Eq. (1.16) we can see that ions are accelerated during the presheath until they reach a velocity $v_B = -\sqrt{k_B T_e/m_i}$ towards the metallic surface. This is usually known as the Bohm criterion, and v_B the Bohm velocity.

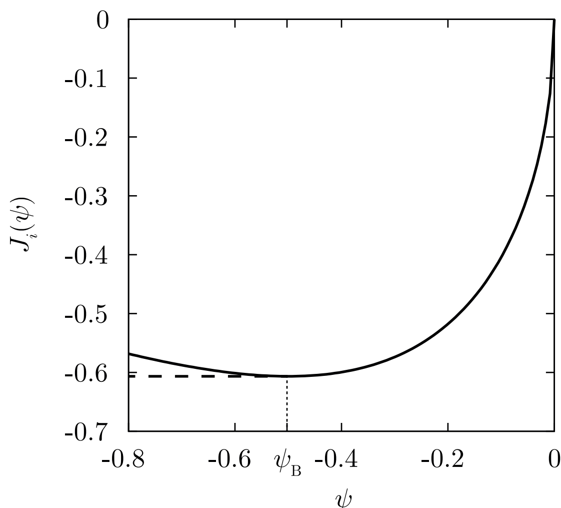


Figure 1.8: Evolution of the ion flux along the presheath as given by Eq. (1.24a) with $\gamma = 1$. Dotted line marks the potential at which J_i is maximum.

The criterion that we have just introduced, establishes the condition that has to be fulfilled in order for the sheath to develop and the potential of the metallic surface to be shielded. On the one hand, this condition can be expressed in terms of the ion velocity. In this case, the Bohm criterion says that the minimum velocity that ions should have, while entering the sheath, in order for the sheath to develop, is the Bohm velocity, *i.e.* $v_i < -\sqrt{k_B T_e/m_i}$. On the other hand, if we look at Eq. (1.27) we see that, when the electric field diverges, the ion flux reaches a maximum. This can be seen in Fig. 1.8, where Eq. (1.24a) has been plotted. There, it is shown that the ion flux reaches a maximum when $\psi = \psi_B$. So, in the zone $\psi_B < \psi < 0$, the presheath provides a mechanism to increase the ion flux from zero, the value at the plasma, to the necessary value at the sheath edge. This ion flux will be the initial condition when solving the sheath equations, and it provides the ion current that the metallic surface is going to drain.

1.4.2. Sheath solution

Now we are going to tackle the problem of finding the solution corresponding to the sheath zone. As we saw in section 1.3, the characteristic length of the sheath zone is the Debye length. So, in order to find the sheath solution we are going to use the variable X as defined in (1.21). Also, we are going to consider the case of low ionisation by taking the limit $\delta \to 0$ in Eqs. (1.22).

$$\frac{d^2\psi(X)}{dX^2} = \frac{1}{2}e^{\psi(X)}\left(1 + \operatorname{erf}\left(\sqrt{\psi(X) - \psi_p}\right)\right) + J_i(X)\sqrt{\frac{\gamma}{-2\psi(X)}} \tag{1.29a}$$

$$\frac{dJ_i(X)}{dX} = 0 \Rightarrow J_i(X) = N_i(X)V_i(X) = N_{is}V_{is} \equiv cte \tag{1.29b}$$

where the subscript s is used to denote the value of variables at the sheath edge, that is, at the starting point of the sheath or the end point of the presheath.

As we can see in Eq. (1.29b), the ion flux is constant along the sheath zone. So, as we introduced in the previous section, the presheath is responsible for providing the ion flux required for the sheath to be formed. Once the end point of the presheath is reached, the ion flux remains constant following the dashed line in Fig. 1.8. This flux gives us the ion current that is drained by the metallic surface per unit area.

Now if we introduce the variables defined in Eqs. (1.21) into the Eq. (1.16) we obtain the following dimensionless energy conservation law:

$$\frac{1}{2}\gamma V_i^2(X) + \psi(X) = \frac{1}{2}\gamma V_{is}^2 + \psi_s = 0 \Rightarrow V_{is} = -\sqrt{-2\psi_s/\gamma} \tag{1.30}$$

On the other hand, the ion density should be equal to the electron density at sheath edge, as quasineutrality has to be fulfilled in the presheath. This leads us to:

$$N_{is} = N_e(\psi_s) = \frac{1}{2}e^{\psi_s}\left(1 + \text{ erf}\left(\sqrt{\psi_s - \psi_p}\right)\right) \tag{1.31}$$

Finally by introducing Eq. (1.31) and Eq. (1.30) into Eq. (1.29b), and then Eq. (1.29b) into Eq. (1.29a) we obtain the Poisson's equation that we have to solve in order to obtain the sheath solution:

$$\frac{d^2\psi(X)}{dX^2} = \frac{1}{2}e^{\psi(X)}\left(1 + \text{ erf}\left(\sqrt{\psi(X) - \psi_p}\right)\right) - \frac{1}{2}e^{\psi_s}\left(1 + \text{ erf}\left(\sqrt{\psi_s - \psi_p}\right)\right)\sqrt{\frac{\psi_s}{\psi(X)}} \tag{1.32}$$

Eq. (1.32) can not be solved analytically, so it has to be solved numerically. Nevertheless, it can be integrated once in order to obtain an expresión for the squared electric field as a function of the potential. If we multiply Eq. (1.32) by $d\psi$, on the left hand side we obtain:

$$\frac{d^2\psi}{dX^2}d\psi = dX\frac{d^2\psi}{dX^2}\frac{d\psi}{dX} = dX\frac{d}{dX}\left[\frac{1}{2}\left(\frac{d\psi}{dX}\right)^2\right] = d\left[\frac{1}{2}(-E)^2\right] = d\left(\frac{E^2}{2}\right) \tag{1.33}$$

which can be integrated directly. The right hand side of Eq. (1.32) can also be integrated, once it is multiplied by $d\psi$. If we perform such integration between any point $(\psi, E(\psi))$ and the sheath edge $(\psi_s, E_s = 0)$, the following solution for the squared electric field is found:

$$\begin{aligned}\left(-\frac{d\psi(X)}{dX}\right)^2 = E^2(\psi) =& \frac{2e^{\psi_p}}{\sqrt{\pi}}\left(\sqrt{\psi_s - \psi_p} - \sqrt{\psi - \psi_p}\right) + e^{\psi}\left(1 + \text{ erf}\left(\sqrt{\psi - \psi_p}\right)\right) - \cdots \\ &\cdots - e^{\psi_s}\left(1 + \text{ erf}\left(\sqrt{\psi_s - \psi_p}\right)\right)\left(1 - 2\left(\psi_s + \sqrt{\psi_s\psi}\right)\right)\end{aligned} \tag{1.34}$$

In Fig. 1.9a it can be seen the right hand side of Eq. (1.34) plotted for different values of the potential at the sheath edge, ψ_s. The biasing potential for the metallic surface has been chosen to be $\psi_p = -5$. In this graph it is shown that, if the potential at the sheath edge is larger than the prediction of the Bohm criterion, $\psi_s > \psi_B = -0.5$, there is a zone where the value of the squared electric field is smaller than zero. This solutions are not physically acceptable, as they would lead to a complex electric field. So, from Eq. (1.34), the Bohm criterion can also be found by imposing that $E^2_{\text{min}} \geq 0$.

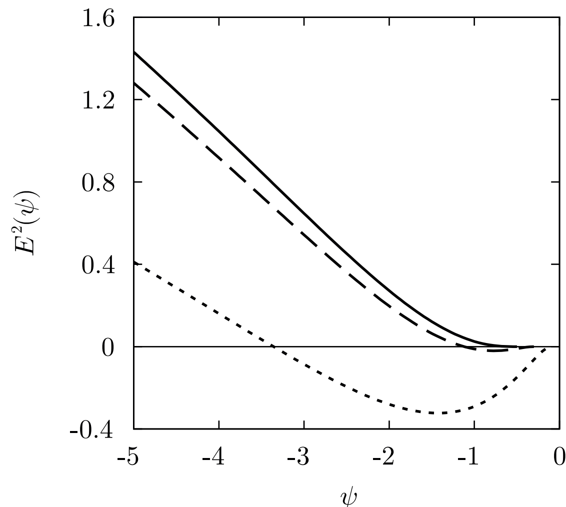
Eq. (1.34) can not be integrated analytically in order to obtain the solution for the potential as a function of the distance to the surface. Also, it is not easy to integrate it numerically for values $\psi_s > \psi_B$, as the numerical method can easily break apart when in the surroundings of the zone where $E^2(\psi) < 0$ in Fig. 1.9a. So, in order to find the potential profile, we have to solve numerically the second order ODE given by Eq. (1.32). This equation can be transformed into the following system of first order ODEs:

$$\begin{cases}\dfrac{d\psi(X)}{dX} = -E(X) \\ \dfrac{dE(X)}{dX} = \dfrac{1}{2}e^{\psi_s}\left(1 + \text{ erf}\left(\sqrt{\psi_s - \psi_p}\right)\right)\sqrt{\dfrac{\psi_s}{\psi(X)}} - \dfrac{1}{2}e^{\psi(X)}\left(1 + \text{ erf}\left(\sqrt{\psi(X) - \psi_p}\right)\right)\end{cases} \tag{1.35}$$

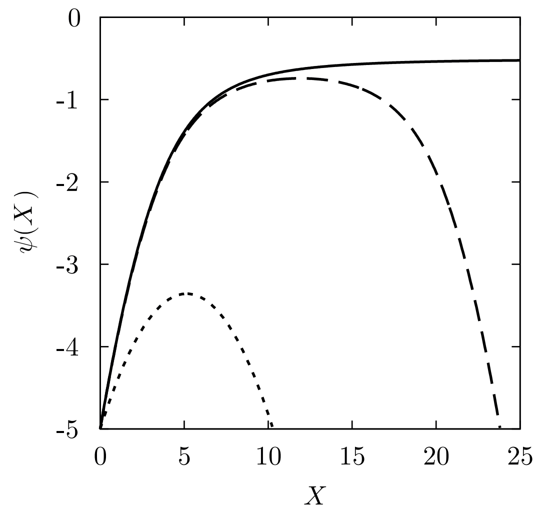
To solve the Cauchy problem associated with the system given by Eqs. (1.35) we need initial conditions for both variables: the potential, and the electric field. If we consider that the metallic surface is located at $X = 0$ and the plasma at some point $X > 0$, our initial condition for the potential would be $\psi(X = 0) = \psi_p$. Also, once we have Eq. (1.34) it is easy to obtain the initial condition for the electric field by substituting $E_p = E(X = 0) = E(\psi = \psi_p)$.

$$E_p = e^{\psi_p}\left(\frac{2}{\sqrt{\pi}}\left(\sqrt{\psi_s - \psi_p}\right) + 1\right) - e^{\psi_s}\left(1 + \text{ erf}\left(\sqrt{\psi_s - \psi_p}\right)\right)\left(1 - 2\left(\psi_s + \sqrt{\psi_s\psi_p}\right)\right) \tag{1.36}$$

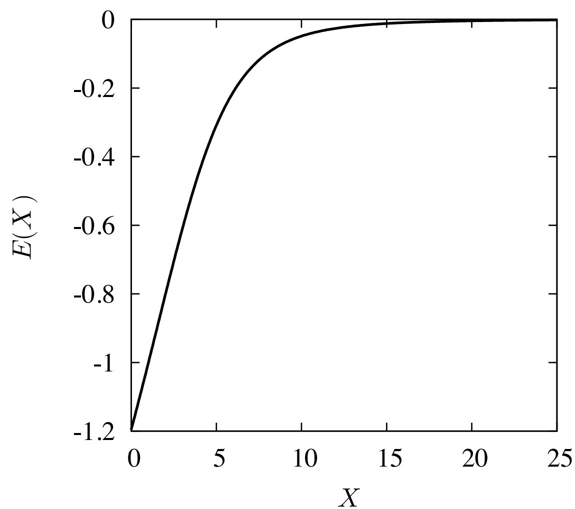
Now the system given by Eqs. (1.35) can be solved numerically. For that task we have used a fourth order Runge-Kutta method (RK4). The results can be seen in Fig. 1.9b. Clearly the initial condition for the electric field given by Eq. (1.36) depends on the potential at the sheath edge ψ_s. In this way we can obtain the different solutions shown in Fig. 1.9b. In this graph we can see again that for a potential at the sheath edge $\psi_s > \psi_B = -0.5$, the solutions found are not physically acceptable. The sheath can not be developed and the perturbation introduced by the metallic surface can not be shielded.



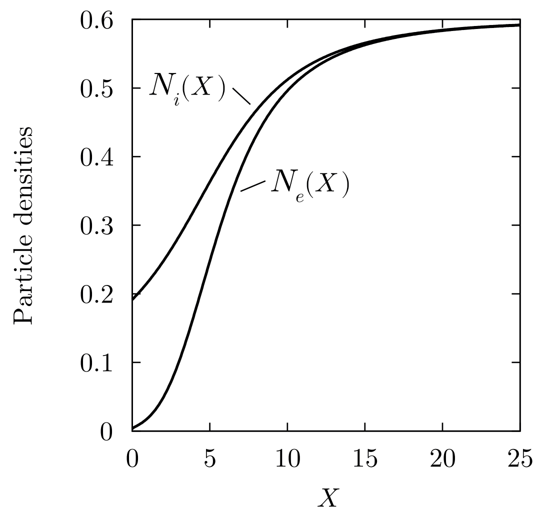
(a) Squared electric field versus the potential as given by Eq. (1.34).



(b) Electric potential distribution obtained by numerical integration (RK4 method) of system (1.35).



(c) Electric field distribution obtained by numerical integration (RK4 method) of system (1.35).



(d) Particle densities distribution obtained by numerical integration (RK4 method) of system (1.35).

Figure 1.9: Solutions of Eq. (1.32) with a biasing potential of the metallic surface $\psi_p = -5$. Sheath edge potential as follows: solid lines $\psi_s = \psi_B = -0.5$, long dashes $\psi_s = -0.4$ and short dashes $\psi_s = -0.1$.

Once the system (1.35) has been solved, it is straightforward to obtain the density profiles for ions and electrons along the sheath. This profiles can be seen in Fig. 1.9d. In this graph we can see how the electrons are repelled and almost none of them reach the metallic surface. The ion density also decreases as we approach the metallic surface, since ions are in free fall, but not as fast as the electron density, so that the net positive charge in the sheath shields out the electric potential of the metallic surface. On the other hand, as we move away from the surface the densities of both species come closer and closer until quasineutrality is reached at the sheath edge. In Fig. 1.9c we can also see, how the electric field becomes negligible at this point.

Finally, we have to notice one more thing from the physically acceptable sheath solution. As can be seen in Fig. 1.9b, the potential doesn't reach the plasma potential. As we increase X we find that

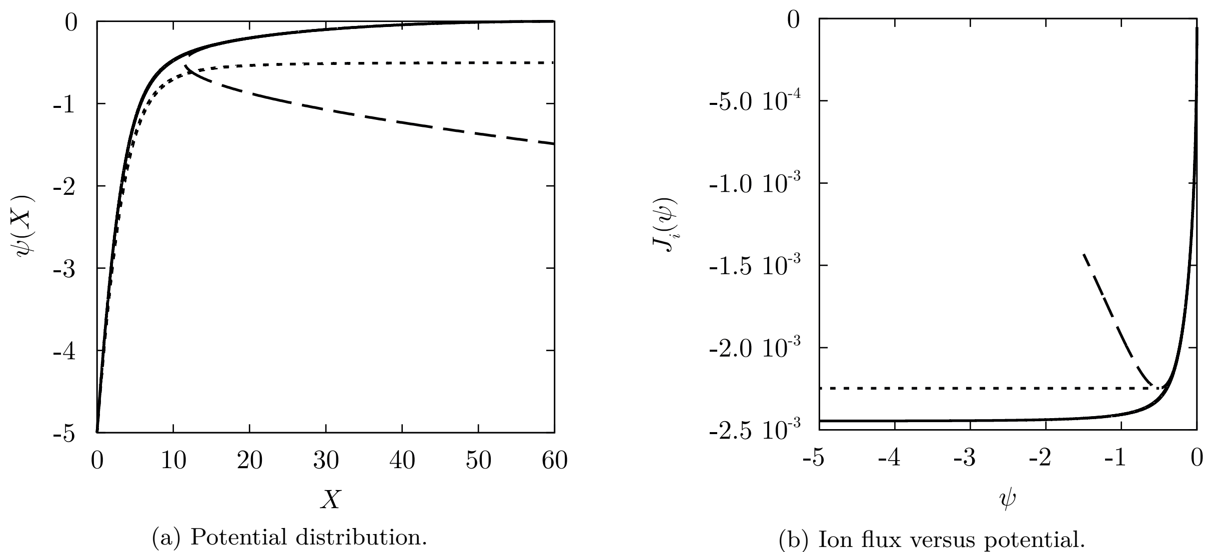
$\psi(X \to \infty) \to \psi_s$. So we can say that, in the length scale of the sheath and in the low ionisation case, the plasma is infinitely far away from the metallic surface and it can never be reached.

1.4.3. Complete solution

From the mathematical point of view, the system of Eqs. (1.23) represents a boundary layer problem. Boundary layer theory is a collection of perturbative method for solving differential equations, or systems, where the highest derivative is multiplied by a perturbing parameter that is very small. In the system (1.23), it is clear that δ is the perturbing parameter. The ionisation rate is very small, in the case we are interested of low pressure plasmas, and we can see that it multiplies the second derivative of the electric potential. Boundary layer theory establishes that there exist narrow regions where the solution of the differential equation changes rapidly. The width of these regions should vanish as the perturbing parameter tends to zero, in our case $\delta \to 0$. Outside the boundary layer, or inner region, we have the outer region, where the solution varies slowly enough in order to neglect the highest order derivative of the problem.

What has been done in the previous sections is finding the approximate solutions corresponding to the outer region and the boundary layer. This solutions are usually referred to as the outer solution, which corresponds to the quasineutral solution, and the boundary layer solution or inner solution, which corresponds to the sheath solution. Theoretically this two solutions can be asymptotically matched in order to find a uniform or complete solution to the problem. However, in our case the solutions found do not fulfil the criteria necessary in order to asymptotically match them.

As it is not possible to obtain the complete solution from the sheath and quasineutral solution, we have to solve the complete system of Eqs. (1.22) with a small but not negligible δ value. This is numerically done with the RK4 method. The solutions of this problem are shown in Fig. 1.10 for some specific δ, γ, ψ_p and ψ_s values.



(a) Potential distribution.

(b) Ion flux versus potential.

Figure 1.10: Solutions of the system (1.22) with $\psi_p = -5$, $\psi_s = -0.5$, $\delta = 5 \times 10^{-5}$ and $\gamma = 72821.0$ (Ar^+). Solid lines correspond to the complete solution found by numerical integration with RK4, long dashes correspond to the quasineutral (outer) solution, and short dashes correspond to the sheath (inner/boundary layer) solution.

In Fig. 1.10a we can see how the potential goes from the values predicted by the quasineutral solution close to the plasma, to the values predicted by the sheath solution near the metallic surface. It has to be noticed that, the value chosen for the ionisation rate, $\delta = 5 \times 10^{-5}$, is small but not negligible. By doing so, it is easier to see clearly the difference between the three solutions shown in Fig. 1.10a. If the δ value is decreased, the transition between solutions turn out to be smoother.

On the other hand, the gap between the ion flux predicted by the complete and the sheath solution that is shown in Fig. 1.10b decreases as $\delta \to 0$. So, in the case of a collisionless plasma, where the

ionisation rate is negligible, the ion flux at the metallic surface can be approximated by the value given by the sheath solution. This fact will be important in the following section where we will introduce the concept of Langmuir probe.

1.5. Langmuir probes and plasma diagnosis

It was back in the 1920's when Irving Langmuir and Harold M. Mott-Smith started their work with gaseous discharges [3–6]. In order to characterise the properties of the different discharges, they used what they called at the time "collectors" [7]. Nowadays, such a device is usually reffered to as a Langmuir probe. A **Langmuir probe** is a metallic electrode that can be biased while inserted in a plasma and that allows us to measure the current that it collects from the plasma.

1.5.1. Different probe types

The geometry of a Langmuir probe and its biasing potential greatly affect the current it collects. The most common geometries of a Langmuir probe are: planar, cylindrical and spherical. While the use of Langmuir probes is widespread as a plasma diagnosing technique [8–14], the actual geometry and dimensions of them, depend on the specific plasma where it is going to be used.

Figure 1.11: One of two spherical Langmuir probes on board ESA's space vehicle Rosetta, due for a comet. Image Credit: ESA/Rosetta Mission/Swedish Institute of Space Physics in Uppsala

For example, in Fig. 1.11 we can see one of the two Langmuir probes that where installed on board the space vehicle of the Rosetta mission [15]. Both probes consist of identical spheres of 2.5 cm in radius mounted on 15 cm "stubs" that are then attached to the main body of the space vehicle. They are made out of titanium and the goldish colour of the probe is due to the titanium nitride (TiN) coating they have. In the case of these probes, the spherical shape and large size of them is explained because of the tenuous plasmas they are designed to analyse. This device is to be used in the interplanetary medium, as well as in the surroundings of the earth, mars [16] and the comet 67 P/Churyomov-Gerasimenko.

Cylindrical Langmuir probes are, probably, the most widely used geometry, specially among laboratory, fusion and industrial plasmas. Mostly because of their ease of manufacturing and theoretical modelling as infinitely long cylindrical conductors. Also, the perturbation introduced by this kind of probes is usually negligible. This is the geometry that this work is centred on, and we will delve into the details of this kind of probes in the following chapter, specifically when they are negatively biased with respect to the plasma.

On the other hand, planar Langmuir probes are not that common. The main disadvantage of them is that it is not easy to neglect boundary effects, in order to model them as an infinite planar conductor. This could be done by making the probe large enough, compared to the Debye length. Nevertheless, this approach is not always acceptable in laboratory and industrial plasmas, as it would most likely cause an equally large perturbation in the plasma that is to be diagnosed, which is an undesirable effect. However, there are situations where this geometry may come in handy, *e. g.* probes designed to be mounted on divertor plates in fusion devices, where the probe has to overcome huge heat fluxes [17]. Anyway, fusion devices are not the only place where planar probes are used [18, 19].

Finally, for the sake of completeness, there are also probes with more complex geometries or configurations, which are usually designed for specific purpose. Among this category of probes we can find ball-pen probes [20], double and triple probes [21], etc. The study of the behaviour of this kind of probes is rather intricate, and rely on parameters such as the magnetic field, the probe orientation, etc.

1.5.2. Current to voltage characteristic curve of a Langmuir probe

The exact shape of the current to voltage ($I - V$) characteristic curve of a Langmuir probe, depends on the properties of the plasma where it is being used, as well as on the geometry and other attributes of the probe itself. Nevertheless, the $I - V$ characteristic curve of any Langmuir probe can be qualitatively described as the one that appears in Fig. 1.12. The diagnosis of a plasma is based on the comparison of the current, in one zone or point of the $I - V$ characteristic curve, with the prediction of a particular theoretical model. In this way, the current and biasing voltage of the probe are correlated to properties of the plasma, as the electron density or temperature.

We have to notice that, in Fig. 1.12, we have taken as positive the current collected by the probe due to electrons, and negative the current collected due to ions, which is a common practice among the plasma diagnosis community. Also, there are two biasing potentials that divide the characteristic into three differentiated zones. The first being the **plasma potential**, ψ_0, which is the potential of the bulk unperturbed plasma. In all our work we consider this potential to be zero, which is equivalent to take the plasma as our reference for the potential. So the biasing potential of the probe, ψ_p, will be always considered with respect to the plasma potential. The second is the **floating potential**, which is the biasing potential of the probe when the net current collected by it is zero. In this case, the number of positive and negative charges that reach the probe per unit of time, *i. e.* the electron and ion currents, must be the same.

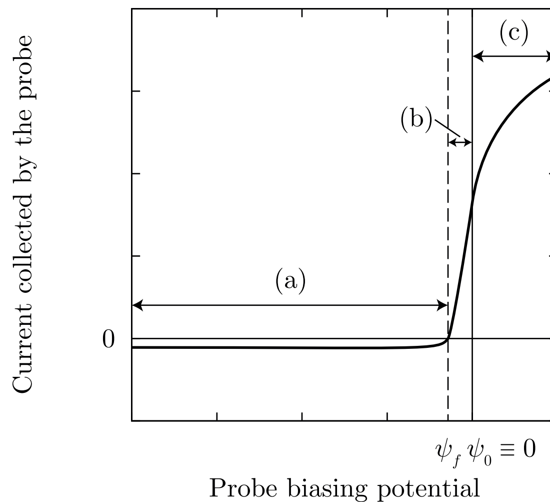


Figure 1.12: Qualitative description of the current to voltage characteristic curve of a Langmuir probe. The biasing potential is measured with respect to the plasma potential and the different zones are: (a) ion saturation, (b) electron retarding and (c) electron saturation.

Now we can establish the three different zones shown in Fig. 1.12. When the biasing potential of the probe is higher than the plasma potential, we are in the electron saturation zone. On the other hand, when the biasing potential is smaller than the plasma potential but higher than the floating potential, we are in the electron retarding zone. And finally, when the biasing potential is smaller than the floating potential, we are in the ion saturation zone. Let us characterise a little bit the different zones that we have just introduced.

(a) Ion saturation zone: When the biasing potential is very negative with respect to the plasma, the probe completely repels all the electrons coming from the plasma. Under this conditions the current collected by the probe is only due to ions. In Fig. 1.12 it can be seen that the current collected in this zone is very small compared to the current collected in other zones of the characteristic curve. As the current drained from the plasma is so small, the perturbation produced in the plasma that is being diagnosed becomes negligible. This is one of the reasons that make the use of this zone particularly interesting in plasma diagnosis, as well as the motivation to center this work in the ion saturation zone.

The reader may have noticed that the model we have solved in section 1.4 corresponds, precisely, to the case of a planar Langmuir probe in the ion saturation zone. As we stated there, in the case of low ionisation ($\delta \to 0$), the ion flux at the surface of the probe can be easily obtained from Eq. (1.29b). So, provided that the probe is large enough to be considered as infinite, the dimensionless ion current collected by it can be expressed as:

$$I_i(\psi_p) = -\frac{A_p}{2} \exp\left(\frac{-1}{2}\right) \left(1 + \operatorname{erf}\left(\sqrt{-\left(\frac{1}{2} + \psi_p\right)}\right)\right) \sqrt{\frac{1}{\gamma}} \tag{1.37}$$

where A_p is the probe area in λ_D^2 units and, obviously, the Bohm criterion has been used to fix the potential at the sheath edge. Also, it has to be noticed that, in addition to the definitions of dimensionless units given in (1.21), in Eq. (1.37) the electric charge is measured in e units.

By analysing Eq. (1.37), it can be seen that, as $\psi_p \to -\infty$, a saturation value is reached because of the asymptotic behaviour of the error function. So, the ion current collected by the probe easily comes to a "saturation" value as long as $\psi_p \lesssim -5$. This saturation current is given by:

$$I_{i\ \mathrm{sat}} = -A_p \sqrt{\frac{1}{\gamma}} \exp\left(\frac{-1}{2}\right) \tag{1.38}$$

To better understand how this expression allows us to diagnose a plasma, we can undo the change of variables given in Eqs. (1.21) to obtain the ion saturation current collected by the probe in SI units. By doing this, we get:

$$I^*_{i\ \mathrm{sat}} = -eA^*_p n_{e0} \sqrt{\frac{k_B T_e}{m_i}} \exp\left(\frac{-1}{2}\right) \tag{1.39}$$

therefore, if we know an approximate value of the electron temperature, we can diagnose the electron density by measuring the ion saturation current

On the other hand, in the case of cylindrical or spherical probes, there are several theoretical models depending on how ions approach the probe. It is this multiplicity of theoretical models which motivates this work, as they provide different values of particle densities and temperatures and experimentalist are not always sure which one should be used. For this reason in the following chapter we will study this models in full detail.

(b) Electron retarding zone: In this zone, the biasing potential is still negative with respect to the plasma. So, as in the previous zone, electrons are repelled by the electrode and ions are attracted by it. Nevertheless, in this case the probe potential is higher than the floating potential, which means that the ion current is overtaken by the electron one. Actually, as the biasing potential is increased over the floating potential, the electron current becomes more and more important and the ion current can be diminished. For this reason, in the electron retarding zone the net current collected by the probe is usually approximated by the electron current.

Because the biasing potential of the probe is still negative with respect to the plasma, the theoretical model to explain the electron current collected in this zone is very similar, if not simpler, than the model developed in section 1.4. If we consider that electrons are in thermal equilibrium in the unperturbed plasma, their distribution function in the plasma is:

$$f(\vec{v}) = n_{e0} \left(\frac{m_e}{2\pi k_B T_e}\right)^{3/2} \exp\left(-\frac{m_e v^2/2}{k_B T_e}\right) \tag{1.40}$$

and the electron current collected by the probe is given by:

$$I^*_e = -eA^*_p \int f_e(\vec{v}) v_\perp d\vec{v} \tag{1.41}$$

where the asterisk means that the electron current and the probe area are to be taken in SI units, $v_\perp$ stands for the component of the velocity perpendicular to the surface of the probe, and the integral has to be performed for all the values of the velocity that allow an electron to reach the probe. It has to be noticed that Eq. (1.40) and (1.41) are independent of the geometry of the probe, so they can be used for the case of planar, cylindrical and spherical probes.

If we consider the planar case, by the opposite argument used in order to obtain Eq. (1.12), v_y and v_z can take any value while $v_x < -\sqrt{-2e\varphi_p/m_e}$. So, Eq. (1.41) ends up like:

$$I^*_e(\varphi_p) = -eA^*_p \int_{-\infty}^{-\sqrt{-2\varphi_p/m_e}} dv_x \int_{-\infty}^{\infty} dv_y \int_{-\infty}^{\infty} dv_z\ f_e(\vec{v}) v_x \tag{1.42}$$

and by direct integration the following expression is easily obtained:

$$I^*_e(\varphi_p) = -eA^*_p n_{e0} \sqrt{\frac{k_B T_e}{2\pi m_e}} \exp\left(\frac{e\varphi_p}{k_B T_e}\right) \tag{1.43}$$

Eq. (1.43) provides one of the oldest methods for diagnosing the electron temperature. If the ion current is not taken into account, by plotting the logarithm of the current collected by the probe,

the electron temperature can be obtained from the slope. Another method for the diagnose of the electron temperature is to measure the floating potential, which in the case of maxwellian electrons can be evaluated as:

$$\varphi_p = -\frac{k_B T_e}{2e} \ln \left(\frac{1}{2\pi\chi^2} \frac{m_i}{m_e} \right) \tag{1.44}$$

where χ is a coefficient around 0.6 [22].

One of the main advantages of this zone is that it can be used to obtain the electron energy distribution function (EEDF). This is done by substituting the velocity distribution function of Eq. (1.41) by its energy counterpart.

(c) Electron saturation zone: Finally, when the biasing potential of the probe is higher than the plasma potential, the probe changes its behaviour and repels positive charges, *i. e.* ions, and attracts negative charges, *i. e.* electrons. If we neglected the ion current in the previous zone, under this conditions, we have more reasons to do the same, so the probe current is completely due to electrons.

The electron current collected by the probe in this zone can be modelled exactly the same that it was done for ions in the ion saturation zone. The only precaution that has to be considered is to interchange the equation of ions and electrons, as now they behave the opposite way. So, all the expressions obtained for the ion saturation zone are the same for this zone, as long as we use the parameters corresponding to the electrons instead of the ion ones and the other way around.

The starting point of this zone is determined by the plasma potential, which can be obtained by finding the inflection point in the $I - V$ characteristic shown in Fig. 1.12.

1.6. Conclusion

In this chapter we have covered the basic fundaments of our research. We have started with the definition of what a plasma is, and what are the properties it should fulfil in order to be considered as a plasma. Later on, we have described the problem of the contact of a metallic surface with a plasma, which allowed us to introduce the sheath, quasineutral and complete solutions. The boundary layer structure of the problem, that was shown there, results in one of the main complexities of the study of the plasma-surface interaction from a fluid perspective. Finally, we have defined what a Langmuir probe is and how its $I - V$ characteristic curve can be used to diagnose a plasma. The different zones in the characteristic curve have been described and a few expressions to evaluate the current drained by the probe have been given.

In the following chapter, our interest will be focused on the study of the ion current collected by a cylindrical Langmuir probe in the ion saturation zone.

Chapter 2

Theories of the ion current collected by a cylindrical probe

2.1. Introduction

When a Langmuir probe is negatively biased with respect to the plasma, it repels negative charges and attracts positive ones. Actually, when the biasing potential is low enough, *i. e.* $\varphi_p \ll -k_B T_e/e$, we enter what is called the ion saturation zone of the $I-V$ characteristic curve of the probe (see Fig. 1.12). Under this conditions, as it has been previously stated, positive ions are the only kind of particle that contribute to the current collected by the probe.

In order to use the $I-V$ characteristic curve to diagnose plasma parameters, a theoretical model that predicts the current collected by the probe is needed. In the ion saturation zone, we only need to know the ion current collected by the probe. In the present chapter we are going to tackle this problem, and during the following sections we will review the different models that predict ion current which have been developed along the history, as well as their range of applicability. Our review is going to be focussed on models of the ion current collected by cylindrical Langmuir probes.

2.2. Ion saturation zone and cylindrical probes. Why?

In the previous chapter we introduced the $I-V$ characteristic curve of a Langmuir probe as well as the different zones in it. Those zones can be seen in Fig. 1.12 along with the two biasing potentials that divide the characteristic: the floating potential, $\psi_f = \varphi_f e/k_B T_e$, and the plasma potential, $\psi_0 = 0$. There, it is highlighted that the main difference between the ion and electron saturation zones of the characteristic curve, is that the absolute value of the current collected is much smaller in the former than the latter. From now on, our study will be centred on the ion saturation zone, that is $\psi_p < \psi_f$, but why?

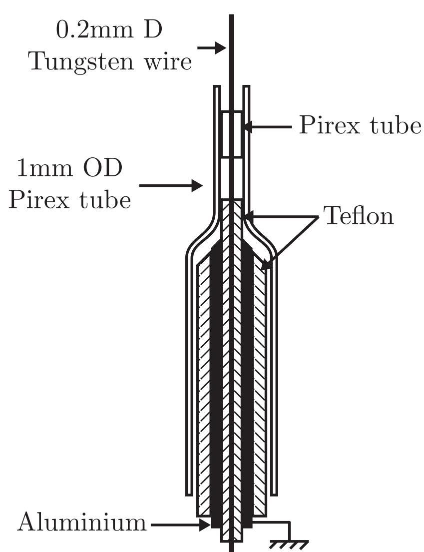
The main reason behind the use of the ion saturation zone for diagnosing a plasma is, precisely, that the current collected in this zone is almost negligible when compared with the current collected in the electron saturation zone of the characteristic. For that reason, the perturbation introduced into the plasmas, as a consequence of the current drained from it by the probe, is minimum. The huge difference between the ion saturation current and the electron one can be explained because of the, also huge, mass ratio between ions and electrons, $m_i/m_e \gg 1$. Let us explain this more in depth.

The current collected by the probe due to each of the species is directly proportional to the product of the number density of particles reaching the probe by the mean velocity with which they reach it, $I \propto n_p v_p$. While the number density of ions and electrons reaching the probe, in their respective saturation zones of the characteristic, is of the same order of magnitud due to the charge neutrality, the mean velocity with which they arrive is not. This velocity can be estimated as $v_p \simeq \sqrt{2E/m}$, where E is the mean energy the particles have when reaching the probe. This energy is, in turn, proportional to the biasing potential of the probe, whose absolute value is also of the same order of magnitud in the ion and electron saturation zones. So, finally, the ion to electron saturation current ratio can be estimated

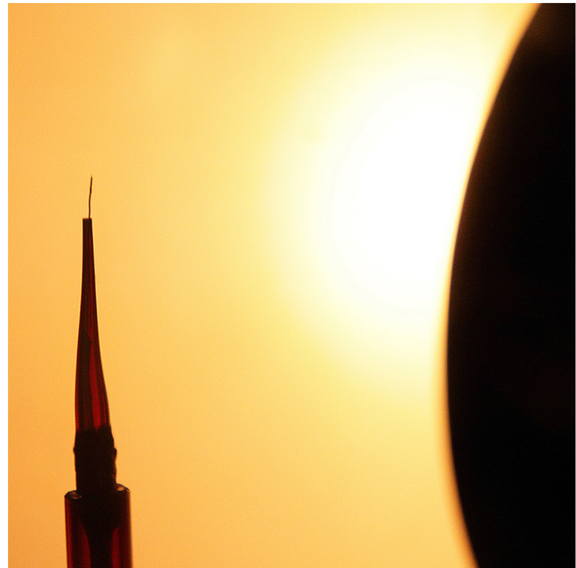
as $I_{e\ \mathrm{sat}}/I_{i\ \mathrm{sat}} \simeq \sqrt{m_i/m_e}$. Once we know that, considering the case of the lightest possible ions, *i. e.* hydrogen ions, the mass ratio would be $m_i/m_e \simeq 2000$ and the saturation current ratio would be $I_{e\ \mathrm{sat}}/I_{i\ \mathrm{sat}} \simeq 45$.

It has to be noticed that, the fact that the ion current is so small, represents a challenge from the experimental point of view. When dealing with such small currents, the noise introduced by the plasma or other components of the experimental setup, could easily overcome the signal which is being measured. For that reason, special care it is needed to be taken in order to maintain the noise floor low enough. Moreover, smoothing techniques [23] could be required prior to the analysis of the experimental data.

The second question to face is, why to center our study in Langmuir probes with cylindrical geometry. Actually, there are two reasons for that. On the one hand, such a geometry is one of the most popular across different kind of plasmas because of their convenience and ease of manufacturing. Besides, this geometry has been and still is extensively used by the research group where I have developed the present study in the laboratories of the University of Córdoba. In Fig. 2.1 it can be seen both, the design and actual implementation of the Langmuir probes we use in our experimental setups.



(a) Design of the probe



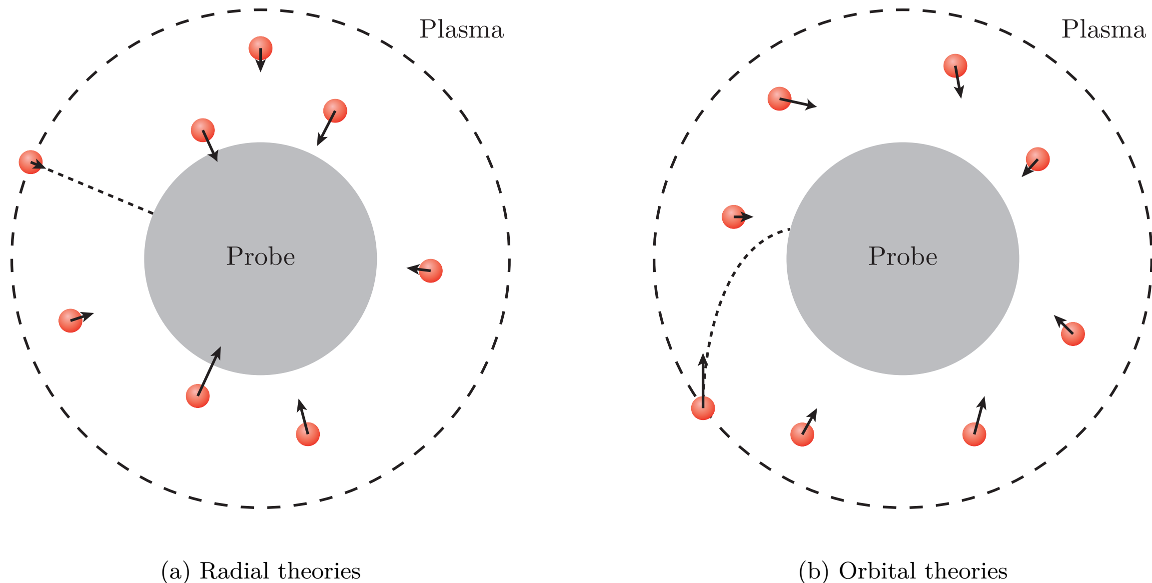
(b) Actual probe inside plasma chamber

Figure 2.1: Cylindrical Langmuir probe used by the Electronegative Plasmas research group of the University of Córdoba. Design and actual implementation of the probe has been carried out by the group. Images credit: Grupo de Plasmas Electronegativos (TEP-230), Universidad de Córdoba.

On the other hand, we are also interested in the cylindrical geometry because of the lack of consensus among researchers about which theoretical model should be used to perform the diagnosis. As we will see in the following sections, there are two main approaches to study the ion current collected by a cylindrical probe. The first approach is to assume that ions fall towards the probe by following orbital trajectories around it, meaning that some of the ions could orbit around the probe and go back to the plasma, and thus, not contributing to the collected current. The other approach is to assume that, positive ions move by following a radial trajectory towards the probe and none of them could orbit it. In Fig. 2.2 the difference between both theories can be seen graphically.

Obviously, depending on the model used to perform the diagnosis, different results are obtained. So, this ambiguity is an important problem that still needs to be addressed in Langmuir probe theory, in order to obtain the proper values of the diagnosed magnitudes. During the past, experimentalist have found measurements in good agreement with both radial and orbital theories and, more important, recently a transition between the two theories has been found. All these facts suggest that, a good theoretical model for the ion current collected by a cylindrical probe, should be somewhere in between the radial and orbital theories, being those two limiting cases recovered under certain conditions.

Finally, let us define a little bit better the boundaries of the ion saturation zone. In Fig. 1.12 we can see that the ion saturation zone starts as soon as $\psi_p < \psi_f$. The value of the floating potential depends on the geometry of the electrode and the plasma conditions, but in most cases and models it ranges between



(a) Radial theories

(b) Orbital theories

Figure 2.2: Graphic visualization of the radial vs orbital probe theories as seen from a cut perpendicular to the axis of a cylindrical probe.

$\psi_f = -3$ and $\psi_f = -4$. But if the biasing potential is just slightly smaller than the floating potential, it is possible that a few electrons reach the probe and contribute to the collected current. This could be seen in Fig. 1.9d for the case of a planar probe, there we can see that the number of electrons reaching the probe is almost but not exactly zero, when the probe potential is $\psi_p = -5$. For that reason, when the diagnose is to be performed in the ion saturation zone, the maximum potential to which the probe is biased must fulfil the condition $\psi_p \ll \psi_f$. Under this condition, we can be sure that the current collected by the probe is completely and exclusively due to ions. So, a typically safe value would be $\psi_p \leq -10$.

We have stablished the maximum value for ψ_p, but what about its minimum value? In the previous chapter we stated that, theoretically, the current collected by the probe should rapidly reach a saturation value as $\psi_p \to -\infty$. Nevertheless, experimentalist find that this saturation value is never reached, and ion current keeps increasing. The problem is that, as the biasing potential becomes more and more negative, the energy with which ions impact the surface of the probe is increased. When ions reach a certain energy at the impact with the probe, secondary emission of electrons from the probe may take place and, obviously, the withdrawal of electrons from the probe computes as ion current. For that reason, when the biasing potential is negative enough, the current due to the secondary emission of electrons appears superimposed over the ion saturation current. From the experimental point of view, typically safe values for the biasing potential, in order to avoid secondary emission current, would be $\psi_p \geq -50$.

We have to notice that, although this work is devoted to the theoretical study of Langmuir probes, we have made the aforementioned remarks about the boundaries of the ion saturation zone in order to clarify some of the working hypotheses of the theories that we are going to discuss. Now, without further ado, let us proceed with the review of the different models of the ion current collected by cylindrical probes. The next section will be devoted to the orbital theories.

2.3. Orbital theories of the ion current collected by a cylindrical probe

We refer to orbital theories as the ones where the orbital motion of ions around the probe is allowed. The first orbital theory of the ion current collected by a probe was developed in 1926 by Harold M. Mott-Smith and Irving Langmuir [7]. This was also the first probe theory at all, and stablished the starting point of the plasma diagnosing techniques with Langmuir probes. The theory was developed for cylindrical and spherical probes. This was a very basic model which only considered conservation laws. Later on, in 1959, Ira B. Bernstein and Irving N. Rabinowitz [24] extended the theory of Mott-Smith and

Langmuir in order to take into account the specific shape of the electric potential distribution across the sheath. They performed a meticulous classification of all the posible trajectories that ions can follow in their fall to the probe, and particularised their model for the case of monoenergetic ions. Finally, in 1967, James G. Laframboise [25] extended again the orbital theory by considering the more realistic case of a fully Maxwellian plasma. He performed the most thorough study about Langmuir probes at the time. He found complete $I-V$ characteristic curves for the cylindrical and spherical cases, for different plasma conditions and probe sizes.

2.3.1. Mott-Smith and Langmuir model

The theory of Mott-Smith and Langmuir was developed in 1926 [7]. Their aim was to obtain the current collected by probes when biased at a certain potential, without taking into account the particular shape of the potential profile between the plasma and the probe. That is why no differentiation were made between sheath and presheath, as we have made in the previous chapter. At that time, it was considered that between the probe and the plasma a sheath needs to be developed, in order to shield the biasing potential of the probe, but at the sheath edge plasma conditions are fulfilled.

Within that framework, Mott-Smith and Langmuir considered a few general assumptions about the plasma and the probe, and developed their theoretical model by considering only conservation laws. The working hypotheses in which the model relies are the following:

- The characteristics of ions and electrons at the sheath edge are known. In particular their densities and distribution functions.
- The smallest mean free path is much larger than the Debye length, that is, collisions do not occur inside the sheath. This condition is usually fulfilled in low pressure and low temperature plasmas.
- The biasing potential of the probe is large compared to the plasma potential.
- The surface of the probe is perfectly absorbent, meaning that any particle that reach it is absorbed. So, particles do not bounce off the surface and no secondary emission occurs.
- The perturbation produced by the probe in the plasma is negligible.

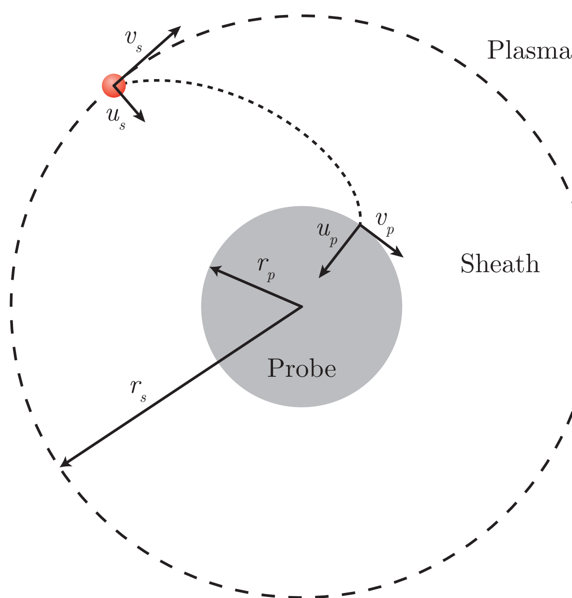


Figure 2.3: Diagram of the Mott-Smith and Langmuir model for a cylindrical probe as seen from above.

In their original work, Mott-Smith and Langmuir studied not only the cylindrical geometry but the spherical and planar cases. Nevertheless, we are going to consider only cylindrical probes. In any case, the reasonings we are going to use can be easily extrapolated to other geometries. One last consideration that has to be made in the case of cylindrical geometry, is that the probe must be long enough, $L \gg r_p$, in order to avoid end corrections and to be able to consider it as an infinite cylinder.

Let us start with the conservation laws that are going to be considered, which are the conservation of energy and the conservation of angular momentum, both for ions. Those laws can be written in the following way:

$$E = \frac{1}{2}m_i(u^2 + v^2 + w^2) + e\varphi(r) = cte \quad (2.1a)$$

$$J = m_i v r = cte \quad (2.1b)$$

u, v and w being the components of the velocity of ions in cylindrical coordinates and r their radial position with respect to the axis of the probe. In Fig. 2.3 a transverse section of the probe can be

seen. There, it is shown that u, v and w are the radial, azimuthal and axial components of the velocity respectively. Also, we are going to denote the values of different magnitudes at the sheath edge by an "s" subscript, and the values at the probe surface by a "p" subscript.

As we said, we are considering that the velocity distribution function for ions normalised to the ion density, $f(u, v, w)$, is known at the sheath edge. That is, $f(u_s, v_s, w_s)du_s dv_s dw_s$ is the average density of ions at the sheath edge with velocities $u_s \in [u_s, u_s + du_s]$, $v_s \in [v_s, v_s + dv_s]$ and $w_s \in [w_s, w_s + dw_s]$. Once we know that, it is clear that the ion flux crossing the sheath edge (dashed line in Fig. 2.3) with velocities in the previously said range, can be evaluated as $f(u_s, v_s, w_s)u_s du_s dv_s dw_s$. So, the ion current that crosses the sheath edge is given by:

$$dI_i = e2\pi r_s L f(u_s, v_s, w_s)u_s du_s dv_s dw_s \tag{2.2}$$

Eq. (2.2) gives the differential current due to ions with $u_s \in [u_s, u_s + du_s]$, $v_s \in [v_s, v_s + dv_s]$ and $w_s \in [w_s, w_s + dw_s]$. To obtain the total current Eq. (2.2) must be integrated with certain limits of integration for the velocity components. As we are interested in the ion current collected by the probe, conditions for ions at the sheath edge that are going to reach the probe should be found.

On the one hand, if an ion at the sheath edge is meant to reach the probe, it must move towards it. So, its radial velocity at the sheath edge must be positive, that is $0 \leq u_s < \infty$. Also, provided that the probe is long enough, ions can have any value of the axial component of the velocity, so, $-\infty < w_s < \infty$. On the other hand, depending on the value of v_s, ions will follow an orbital trajectory that ends up at the surface of the probe, as the one shown in Fig. 2.3, or they will orbit around it and go back into the plasma. In order to find the condition that v_s must fulfil, we are going to rewrite Eqs. (2.1) between the sheath edge and the surface of the probe:

$$\frac{1}{2}m_i(u_s^2 + v_s^2 + w_s^2) + e\varphi_s = \frac{1}{2}m_i(u_p^2 + v_p^2 + w_p^2) + e\varphi_p \tag{2.3a}$$

$$m_i v_s r_s = m_i v_p r_p \tag{2.3b}$$

where φ_p is the biasing potential of the probe and φ_s is the potential at the sheath edge. As the plasma conditions must be recovered at the sheath edge, φ_s is equal to the plasma potential, which we are going to take as a reference potential, so $\varphi_s = \varphi_0 = 0$. Also, as the electric potential only depends on the radial distance to the probe, there are no forces on the axial direction, so $w_s = w_p$. Now, Eqs. (2.3) can be transformed into:

$$u_p^2 = u_s^2 + v_s^2 - v_p^2 - \frac{2e\varphi_p}{m_i} \tag{2.4a}$$

$$v_p = v_s \frac{r_s}{r_p} \tag{2.4b}$$

Now, if we take Eq. (2.4b) into Eq. (2.4a) and solve for u_p^2 we have:

$$u_p^2 = u_s^2 + v_s^2 \left(1 - \frac{r_s^2}{r_p^2}\right) - \frac{2e\varphi_p}{m_i} \tag{2.5}$$

We know that, in order for an ion to reach the surface of the probe, the condition $u_p \geq 0$ must be fulfilled. So, Eq. (2.5) gives us:

$$u_s^2 + v_s^2 \left(1 - \frac{r_s^2}{r_p^2}\right) - \frac{2e\varphi_p}{m_i} \geq 0 \tag{2.6}$$

Finally, by solving Eq. (2.6) for v_s we can obtain the following integration limits:

$$0 \leq u_s < \infty \; ; \quad -\infty < w_s < \infty \; ; \quad -\underbrace{\sqrt{\frac{r_p^2}{r_s^2 - r_p^2}\left(u_s^2 - \frac{2e\varphi_p}{m_i}\right)}}_{= v_0} \leq v_s \leq \underbrace{\sqrt{\frac{r_p^2}{r_s^2 - r_p^2}\left(u_s^2 - \frac{2e\varphi_p}{m_i}\right)}}_{= v_0} \tag{2.7}$$

Once we have the proper integration limits, we can integrate Eq. (2.2) in order to obtain the ion current collected by the probe as:

$$I_i = e2\pi r_s L \int_0^\infty du_s \int_{-v_0}^{v_0} dv_s \int_{-\infty}^{\infty} dw_s f(u_s, v_s, w_s)u_s \tag{2.8}$$

where $v_0 = \sqrt{\frac{r_p^2}{r_s^2 - r_p^2}\left(u_s^2 - \frac{2e\varphi_p}{m_i}\right)}$.

In their original study, Mott-Smith and Langmuir solved Eq. (2.8) for different distribution functions, including monoenergetic, Maxwellian and Maxwellian with a drift velocity. Nevertheless, here were are going to solve it for the most relevant case of Maxwellian ions. If Maxwellian ions are assumed at the sheath edge, then:

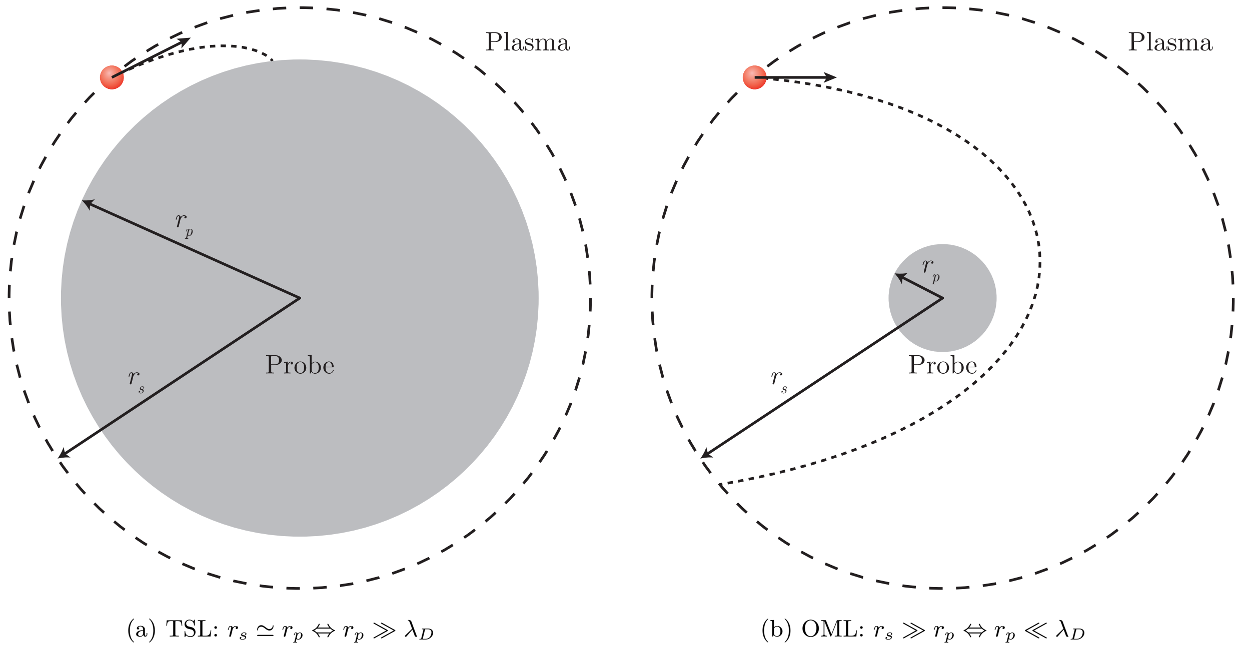
$$f(u_s, v_s, w_s) = n_{is}\left(\frac{m_i}{2\pi k_B T_i}\right)^{3/2} \exp\left(-\frac{m_i}{2k_B T_i}(u_s^2 + v_s^2 + w_s^2)\right) \tag{2.9}$$

n_{is} being the ion density at the sheath edge, which is supposed to be the same that the ion density at the plasma, $n_{is} = n_{i0}$. Also, T_i is the ion temperature at the plasma. By taking Eq. (2.9) into Eq. (2.8) and integrating, yields:

$$\begin{aligned} I_i = e2\pi r_p L n_{i0}\sqrt{\frac{k_B T_i}{2\pi m_i}} \Bigg[& \exp\left(-\frac{e\varphi_p}{k_B T_i}\right) \operatorname{erfc}\left(\sqrt{-\frac{r_s^2}{r_s^2 - r_p^2}\frac{e\varphi_p}{k_B T_i}}\right) \\ & + \frac{r_s}{r_p}\left(1 - \operatorname{erfc}\left(\sqrt{-\frac{r_p^2}{r_s^2 - r_p^2}\frac{e\varphi_p}{k_B T_i}}\right)\right)\Bigg] \end{aligned} \tag{2.10}$$

where, $\operatorname{erfc}(x) = \frac{2}{\sqrt{\pi}}\int_x^\infty e^{-t^2}dt$, is the complementary error function.

Eq. (2.10) gives us finally the ion current collected by a probe as a function of its biasing potential. The problem is that this expression is not very useful, as it depends on the position where the sheath edge is located, r_s, which is a parameter that is not known. For this reason, usually, two limiting cases are considered instead of the general equation. Those two limiting cases, which can be schematically shown in Fig. 2.4, are known as Thin Sheath Limit (TSL) and Orbital Motion Limit (OML). In them, the size of the sheath is considered either, very small or large, when compared to the radius of the probe.



(a) TSL: $r_s \simeq r_p \Leftrightarrow r_p \gg \lambda_D$

(b) OML: $r_s \gg r_p \Leftrightarrow r_p \ll \lambda_D$

Figure 2.4: Graphic visualization of the Thin Sheath Limit (TSL) and Orbital Motion Limit (OML) of Eq. (2.10).

Now, let us discuss briefly the two approximations of the Mott-Smith and Langmuir theory.

Thin Sheath Limit (TSL)

In this case, the sheath surrounding the probe is considered very small compared to the probe radius. As can be seen in Fig. 2.4a, this implies that the position where the sheath edge is

located is comparable to the position of the surface of the probe, $r_s \simeq r_p$. Also, even though we do not know the actual size of the sheath, and consequently the position of the sheath edge, we do know that its order of magnitude is several times the Debye length. So, the usual condition imposed when considering the TSL is $r_p \gg \lambda_D$. Under this condition, it is clear that $r_s^2 - r_p^2 \simeq 0$ and $r_s/r_p \simeq 1$. Introducing these approximations into Eq. (2.10), we observe that the arguments of the complementary error functions go to infinity, so $\operatorname{erfc}(x \to \infty) \simeq 0$, obtaining the current collected by the probe in the TSL as:

$$\boxed{I_i = e2\pi r_p L n_{i0}\sqrt{\frac{k_B T_i}{2\pi m_i}}} \tag{2.11}$$

If we analyse Eq. (2.11), we can see that this current is independent of the biasing potential of the probe, and corresponds to the current due to the thermal flux of ions that cross the sheath edge. This is reasonable, as the small size of the sheath implies that all the particles that cross the sheath edge are going to reach the probe (see Fig. 2.4a).

Orbital Motion Limit (OML)

In this case, contrary to the TSL, the length of the sheath surrounding the probe is considered much larger than its radius. Also, it can be seen in Fig. 2.4b that this situation implies that the positions of the sheath edge and the surface of the probe verify the condition $r_s \gg r_p$. For the same reasons given in the TSL, the previous condition is equivalent to $r_p \ll \lambda_D$. So, under this approximation, it can be assumed that $r_s^2 - r_p^2 \simeq r_s^2$, and Eq. (2.10) can be transformed into:

$$\begin{aligned} I_i = e2\pi r_p L n_{i0}\sqrt{\frac{k_B T_i}{2\pi m_i}} \Bigg[& \exp\left(-\frac{e\varphi_p}{k_B T_i}\right) \operatorname{erfc}\left(\sqrt{-\frac{e\varphi_p}{k_B T_i}}\right) \\ & + \frac{r_s}{r_p}\left(1 - \operatorname{erfc}\left(\sqrt{-\frac{r_p^2}{r_s^2}\frac{e\varphi_p}{k_B T_i}}\right)\right)\Bigg] \end{aligned} \tag{2.12}$$

Eq. (2.12) is simpler that Eq. (2.10) but still depends on the parameter r_s, so we need to take the simplification one step further. Let us consider the Taylor expansion of the complementary error function about $x = 0$:

$$\operatorname{erfc}(x \to 0) = 1 - \frac{2x}{\sqrt{\pi}} + \mathcal{O}\left(x^3\right) \tag{2.13}$$

Now, in the OML we have that $r_p^2/r_s^2 \to 0$, so the second complementary error function that appears in Eq. (2.12) can be expanded by using Eq. (2.13) in order to obtain:

$$I_i = e2\pi r_p L n_{i0}\sqrt{\frac{k_B T_i}{2\pi m_i}} \left[\exp\left(-\frac{e\varphi_p}{k_B T_i}\right) \operatorname{erfc}\left(\sqrt{-\frac{e\varphi_p}{k_B T_i}}\right) + \frac{2}{\sqrt{\pi}}\sqrt{-\frac{e\varphi_p}{k_B T_i}} \right] \tag{2.14}$$

Eq. (2.14) gives us an analytical expression for the ion current collected by the probe in the OML that is independent of the position of the sheath edge. The problem with this expression is that it is not very practical, due to the complex dependence of I_i on φ_p, when it comes to fit experimental data in order to diagnose a plasma. For this reason a simplified version of Eq. (2.14) is desirable. In order to obtain such a simplified versión we consider the, otherwise usual, case of $|e\varphi_p| \geq 2k_B T_i$. When under this conditions, the following asymptotical series expansion of the complementary error function [26] can be used:

$$\operatorname{erfc}(x \neq 0) = e^{-x^2}\left[\frac{1}{\sqrt{\pi}}\frac{1}{x} + \mathcal{O}\left(\frac{1}{x^3}\right)\right] \tag{2.15}$$

and by introducing this approximation into Eq. (2.14) we get:

$$I_i = e2\pi r_p L n_{i0}\sqrt{\frac{k_B T_i}{2\pi m_i}} \left[\frac{1}{\sqrt{\pi}}\frac{1}{\sqrt{-\frac{e\varphi_p}{k_B T_i}}} + \frac{2}{\sqrt{\pi}}\sqrt{-\frac{e\varphi_p}{k_B T_i}} \right] \tag{2.16}$$

Now, if Eq. (2.16) is squared:

$$I_i^2 = \left(e2\pi r_p L n_{i0}\sqrt{\frac{k_B T_i}{2\pi m_i}}\right)^2 \left[\frac{1}{\pi}\overbrace{\frac{1}{\frac{e\varphi_p}{k_B T_i}}}^{\simeq 0} + \frac{4}{\pi}\left(-\frac{e\varphi_p}{k_B T_i}\right) + \frac{4}{\pi}\right] \tag{2.17}$$

And finally, we can obtain the expression that is commonly used when diagnosing a plasma with the OML theory. We have to notice that, contrary to the TSL, now the current depends on the biasing potential of the probe.

$$\boxed{I_i = 4er_p L n_{i0}\sqrt{\frac{k_B T_i}{2m_i}}\sqrt{1-\frac{e\varphi_p}{k_B T_i}}} \tag{2.18}$$

As it is obvious from Eq. (2.18), the diagnose with this theory is performed by a linear fit of the I_i^2 versus φ_p data.

The model of Mott-Smith and Langmuir, even though being the first to be developed and rather simple, it retains most of the physics of the problem and produces results in reasonable agreement when the required conditions are met. As mentioned before, it was also developed for different geometries and distribution functions. However, there are two main problems that this model presents.

The first is that $I-V$ characteristic curves are known to depend on the electron temperature even in the ion saturation zone, while the expressions found by Mott-Smith and Langmuir do not. This deficiency was partially solved by a subsequent model proposed by Bhom, Burhop and Massey in 1949 [27]. There, Bhom *et al.* developed a model of a spherical probe in the TSL approximation considering the influence of electrons. Nevertheless, because of the TSL approximation, their results lacked the dependence of the current on the biasing potential of the probe.

The second problem is that the current collected by the probe is assumed to be independent of the specific shape of the electric potential distribution across the sheath. The solution for this problem was firstly approached by Ira B. Bernstein and Irving N. Rabinowitz, whose theory we are going to review next.

2.3.2. Bernstein and Rabinowitz model

Bernstein and Rabinowitz developed their model in 1959 [24]. They basically extended the theory of Mott-Smith and Langmuir by considering the shape of the electric potential across the sheath. In order to do that, they solved the appropriate Boltzmann's equation, yielding the particle density and flux as functionals of the electrostatic potential profile. Once the particle density is known, the electric potential can be found by using Poisson's equation. They solved this problem for spherical and cylindrical geometries, and for ions with a general distribution function, $f(\vec{v})$, at the plasma. They also considered ions with any degree of ionisation. Finally, they particularised their result for a monoenergetic distribution function for ions, and obtained the corresponding $I-V$ characteristic curves numerically. Here we are going to restrict our review for the case of cylindrical geometry as well as singly ionised ions.

The model of Bernstein and Rabinowitz is not that much different than the one developed by Mott-Smith and Langmuir. They started with pretty much the same working hypotheses and stated that the problem of determining the probe characteristic can be reduced to solve the collisionless Boltzmann's equation:

$$\frac{df}{dt} = \vec{v}\cdot\vec{\nabla} f - \frac{e}{m_i}\vec{\nabla}\varphi\cdot\vec{\nabla}_v f = 0 \tag{2.19}$$

along with Poisson's equation:

$$\nabla^2\varphi = \frac{-e}{\varepsilon_0}(n_i - n_e) \tag{2.20}$$

strictly in the region $r_p < r < r_s$. Where r_p and r_s are taken from the notation previously used. Also, $f \equiv f(\vec{r},\vec{v})$, is the density distribution function in position and velocity space for ions. Eq. (2.19) states that, in the absence of collisions, the distribution function is constant along an ion trajectory in the phase space. The ion number density and flux can be evaluated at any point by performing the following

integrals with proper integration limits:

$$n_i(\vec{r}) = \int d^3\vec{v}\ f \tag{2.21}$$

$$\vec{\Gamma}(\vec{r}) = \int d^3\vec{v}\ f\ \vec{v} \tag{2.22}$$

Although Eq. (2.21) and Eq. (2.22) represent the same approach that we used in the model of Mott-Smith and Langmuir, the difference in the Bernstein and Rabinowitz theory lies in the way the different trajectories of ions are analysed, considering the shape of the potential in order to find the limits of integration. Let us start, like in the previous model, by considering the conservation laws of energy and angular momentum around the axis of the probe:

$$E_\perp = \frac{1}{2}m_i(u^2+v^2) + e\varphi(r) \tag{2.23a}$$

$$E_\parallel = \frac{1}{2}m_i w^2 \tag{2.23b}$$

$$J = m_i v r \tag{2.23c}$$

It has to be noticed that Eqs. (2.23) are the same than Eqs. (2.1), apart from the fact that Eq. (2.1a) has been split into Eq. (2.23a) and Eq. (2.23b), in order to isolate the axial motion of ions along the axis of the probe. Now, if we invert Eqs. (2.23) we get:

$$u = \pm\left[\frac{2}{m_i}(E_\perp - e\varphi(r)) - \frac{J^2}{m_i^2 r^2}\right]^{1/2} \tag{2.24a}$$

$$w = \pm\left[\frac{2E_\parallel}{m_i}\right]^{1/2} \tag{2.24b}$$

$$v = \frac{J}{m_i r} \tag{2.24c}$$

Eqs. (2.24) allow us to write the distribution function for ions as a function of the three constants of motion defined in Eqs. (2.23), that is $f \equiv f(E_\perp, E_\parallel, J)$, instead of $f(u, v, w)$. This ostensibly simple change of variables in the distribution function, allows us to find the proper limits of integration for Eq. (2.21) and Eq. (2.22) in terms of the energy and angular momentum of ions as well as the electric potential. But, before analysing the orbits of ions in order to define the aforementioned limits of integration, we shall notice that the distribution $f(E_\perp, E_\parallel, J)$ can be decomposed into four terms as:

$$f(E_\perp, E_\parallel, J) = f^+(E_\perp, E_\parallel, J) + f^-(E_\perp, E_\parallel, J) + f^\dagger(E_\perp, E_\parallel, J) + f^\ddagger(E_\perp, E_\parallel, J) \tag{2.25}$$

where f^+ corresponds to $u > 0$, f^- to $u < 0$, $f^\dagger$ to $w > 0$ and $f^\ddagger$ to $w < 0$. Nevertheless, because of symmetry reasons, the distribution function is not sensitive to the sing of the axial velocity, w. So, the decomposition of $f(E_\perp, E_\parallel, J)$ can be simplified as:

$$f(E_\perp, E_\parallel, J) = f^+(E_\perp, E_\parallel, J) + f^-(E_\perp, E_\parallel, J) \tag{2.26}$$

On the other hand, as the probe is assumed to be perfectly absorbing, there are ions approaching the probe, that is with $u > 0$, that never go back into the plasma with $u < 0$. So, in general, $f^+(E_\perp, E_\parallel, J) \neq f^-(E_\perp, E_\parallel, J)$.

The next step that Bernstein and Rabinowitz took was to carefully classify all the posible orbits that ions can follow. This analysis was carried out in terms of the effective potential energy:

$$U(r, J) = e\varphi(r) + \frac{J^2}{2m_i r^2} \tag{2.27}$$

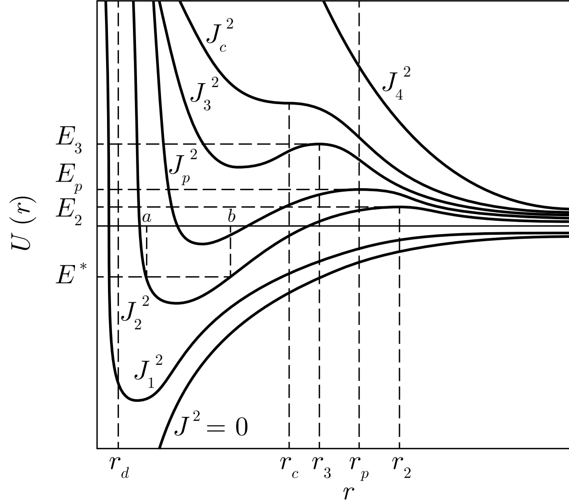
which governs the radial motion of ions. In Fig. 2.5a the dependence of this effective potential on the radial distance to the axis of the probe, r, can be seen schematically for different J^2 values. The curves shown in Fig. 2.5a have been drawn according to the following assumptions:

1. For small r values, $r^2\varphi(r)$ tends to zero as r decreases. This implies that for small r the centrifugal term dominates over the electric potential.

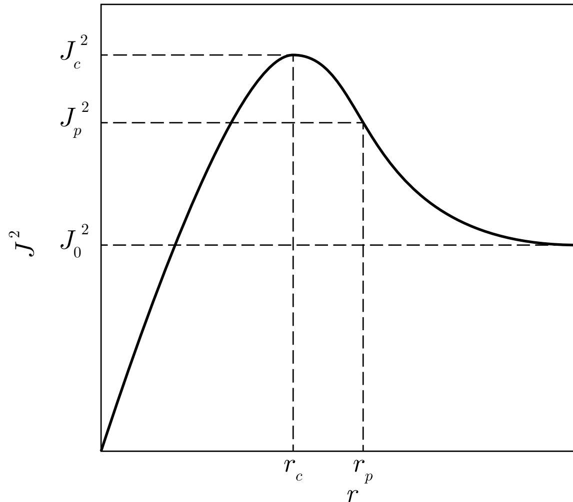
2. For large r values, the asymptotic behaviour of the potential is:

$$\varphi(r \to \infty) \sim \frac{cte}{r^2} = -\frac{J_0^2}{2m_i r^2 e} \tag{2.28}$$

which determines the asymptotic behaviour of the curves $U(r, J)$.



(a) Diagram of the effective potential energy of ions as given by Eq. (2.27).



(b) Diagram of the location of the extrema of U as given by Eq. (2.29)

Figure 2.5: Classification of orbits in term of the effective potential energy of ions around a cylindrical probe

The qualitative behaviour of the effective potential can be readily seen in Fig. 2.5a. For large values of the angular momentum, the centrifugal term, which decreases monotonically with r, dominates over the electric potential everywhere, which increases monotonically with r. So, for large J^2 values, the associated effective potential, U, decreases monotonically to zero. In Fig. 2.5a it can also be seen that when J^2 decreases, the new U curve lies below the old one. This fact can be probed by observing Eq. (2.27), where we can see that $(\partial U/\partial J^2)_r \geq 0$. The effective potential curves retain their monotonic decreasing character until a critical value J_c^2 is reached for the angular momentum. If a slightly smaller J^2 value is considered, the U curves exhibit an inflection point with a minimum to its left and a maximum at some larger r value. Finally, if the J^2 value is further decreased and drops below J_0^2, as given by Eq. (2.28), the behaviour of U for larger r values is ruled by the electric potential, and the curves exhibit only a minimum, past which they increase monotonically to zero.

The location of the extrema of the effective potential can be found by solving the equation $(\partial U/\partial r)_J = 0$. Or, by considering Eq. (2.27):

$$J^2 = m_i r^3 e \frac{d\varphi(r)}{dr} \tag{2.29}$$

Eq. (2.29) can also be used to determine a critical radius, r_c, such that if the probe radius is smaller than it (*i. e.* $r = r_d$ as shown in Fig. 2.5a), then there exist orbits which do not cut the surface of the probe and are radially bounded. For instance, in Fig. 2.5a one of those orbits is shown, it is the trajectory corresponding to a particle with angular momentum J_2 and energy $E^* = U(a, J_2) = U(b, J_2)$, where a and b represent the turning points of the trajectory. However infrequent, collisions determine the population of ions on such orbits, thus this quantity is not easy to evaluate. For this reason, in order to simplify the problem, probe radii greater than r_c are only to be considered. By assuming $r_p \geq r_c$ the possibility of the existence of such trapped ions is avoided, since all the extrema found to the right of r_c are relative maximums (see Fig. 2.5b) and thus do not allow such orbits.

Fig. 2.5b shows the definition of $J_p^2 = m_i r_p^3 e (d\varphi(r)/dr)_{r_p}$, which is the square of the angular momentum associated with the effective potential curve which has its maximum precisely at $r = r_p$. Then,

ions with angular momentum $J^2 \geq J_p^2$ will be absorbed by the probe if $E_\perp \geq U(r_p, J)$, otherwise they will be reflected at some larger radius. On the other hand, if their angular momentum is $J^2 < J_p^2$, they will be absorbed if $E_\perp$ is greater than the maximum value of U, which lies at some $r > r_p$. We have just stablished graphically the limits of integration needed in order to evaluate Eq. (2.21) and Eq. (2.22), but, let us define them in a more mathematical fashion. We are going to start by defining the curve $J^2 = G(E_\perp)$ as a function in the $J^2 - E_\perp$ phase space.

$$J^2 = G(E_\perp) = \begin{cases} 2m_i r_p^2 (E_\perp - e\varphi(r_p)) \; ; & J^2 \geq J_p^2 \\ \left. \begin{array}{l} 2m_i r^2 (E_\perp - e\varphi(r)) \\ J^2 = m_i r^3 e \dfrac{d\varphi(r)}{dr} \end{array} \right\} ; & J^2 < J_p^2 \end{cases} \tag{2.30}$$

In Fig. 2.6 a sketched diagram of the function $J^2 = G(E_\perp)$ can be seen. There, the white zone, denoted as A, corresponds to orbits where ions are absorbed by the probe, as their value of $E_\perp$ is larger than, either the maximum of the effective potential or the effective potential at the surface of the probe. For this reason, in the A zone, we have that $f^+(E_\perp, E_\parallel, J) = 0$ and $f^-(E_\perp, E_\parallel, J) = f(E_\perp, E_\parallel, J)$.

Now, at any point, r, the radial kinetic energy $m_i u^2/2 = E_\perp - U(r, J) \geq 0$. The straight line $E_\perp = U(r, J) = e\varphi(r) + J^2/2m_i r^2$ can also be seen in Fig. 2.6. Obviously, the zone to the right of it, denoted as B, is excluded. Ions with orbits which fall into the B zone would lead to a complex radial velocity. So, in the B zone we have that $f(E_\perp, E_\parallel, J) = 0$. The line $E_\perp = U(r, J^2)$ is tangent to the curve $J^2 = G(E_\perp)$ at the point where $J^2 = m_i r^3 e(d\varphi(r)/dr)$, which is the point of the maximum in the corresponding effective potential curve.

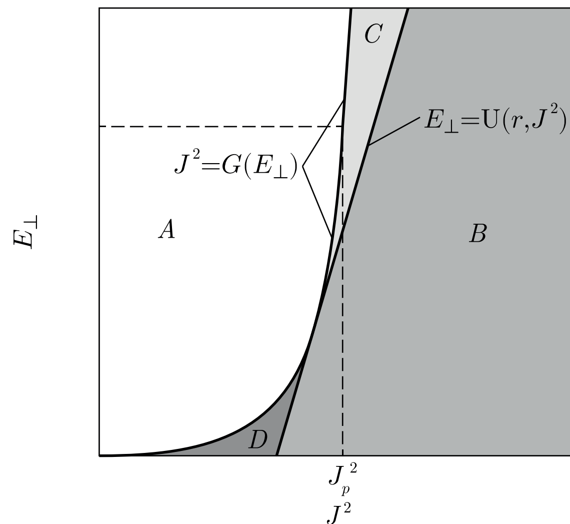


Figure 2.6: Diagram of the limits of integration for Eq. (2.20) and Eq. (2.21) in the E, J^2 phase space.

Finally, there are two more zones that we need to analyse, both of which correspond to orbits that have a turning point to the left or to the right of the maximum of the effective potential curve. The zone denoted as D corresponds to ions with orbits which have a turning point to the left of this maximum. Obviously such ions should come from the probe, but as we have previously stated, the probe is perfectly absorbent, so it is not possible for such ions to exist. For this reason, the D zone is also excluded and $f(E_\perp, E_\parallel, J) = 0$ in it. On the other hand, the zone denoted as C corresponds to ions with orbits that have a turning point to the right of the maximum of the $U(r, J)$ curve. Such ions come from $r \to \infty$ and reach a turning point at a position $r > r_p$. This kind of ions approach the probe but never reach it, instead they go back into the plasma, and for this reason $f^+(E_\perp, E_\parallel, J) = f^-(E_\perp, E_\parallel, J) = \frac{1}{2} f(E_\perp, E_\parallel, J)$ inside the C zone.

Now that we have classified all the posible orbits of ions, we can proceed to the integration of Eq. (2.21) and Eq. (2.22). We shall notice that, in the integration of Eq. (2.22) we are only interested in particles that reach the probe, as we want to evaluate the current collected by it. So, the integration of Eq. (2.22) must be restricted to the zone A. Contrary, in the integration of Eq. (2.21) we are interested in the density of ions at any point $r \geq r_p$ in order to introduce that result into Poisson's equation, so, the integration of Eq. (2.21) must be performed in the zones A and C.

Before carrying out the integration, it has to be noticed that, after the change of variables $(u, v, w) \to (E_\perp, E_\parallel, J)$, the Jacobian of the transformation is needed. This is easily evaluated from Eq. (2.24) and yields the result shown in Eq. (2.31).

$$\mathcal{J}(E_\perp, E_\parallel, J) = \begin{vmatrix} \frac{\partial u}{\partial E_\perp} & \frac{\partial u}{\partial E_\parallel} & \frac{\partial u}{\partial J} \\ \frac{\partial v}{\partial E_\perp} & \frac{\partial v}{\partial E_\parallel} & \frac{\partial v}{\partial J} \\ \frac{\partial w}{\partial E_\perp} & \frac{\partial w}{\partial E_\parallel} & \frac{\partial w}{\partial J} \end{vmatrix} = \frac{1}{\sqrt{2m_i E_\parallel}} \frac{1}{m_i \sqrt{2m_i r^2 (E_\perp - e\varphi(r)) - J^2}} \tag{2.31}$$

Also, it is plausible to assume that the particle distribution functions are independent of the sing of v and w. By taking into account this last consideration, along with Eq. (2.31), we can write Eq. (2.21) as:

$$\begin{aligned} n_i(r) = 2 \int_0^\infty dE_\parallel \int_0^\infty dE_\perp \int_0^{\sqrt{G(E_\perp)}} dJ \\ \cdot \frac{f^-(E_\perp, E_\parallel, J)}{\sqrt{2m_i E_\parallel} m_i \sqrt{2m_i r^2 (E_\perp - e\varphi(r)) - J^2}} \\ + 2 \int_0^\infty dE_\parallel \int_{e\varphi(r)+(er/2)(d\varphi(r)/dr)}^\infty dE_\perp \int_{\sqrt{G(E_\perp)}}^{\sqrt{2m_i r^2 (E_\perp - e\varphi(r))}} dJ \\ \cdot \frac{2f^-(E_\perp, E_\parallel, J)}{\sqrt{2m_i E_\parallel} m_i \sqrt{2m_i r^2 (E_\perp - e\varphi(r)) - J^2}} \end{aligned} \tag{2.32}$$

and Eq. (2.22) as:

$$\vec{\Gamma}(r) = -\vec{e_r} 2 \frac{1}{m_i^2 r} \int_0^\infty dE_\parallel \int_0^\infty dE_\perp \int_0^{\sqrt{G(E_\perp)}} dJ \frac{2f^-(E_\perp, E_\parallel, J)}{\sqrt{2m_i E_\parallel}} \tag{2.33}$$

Once they had the, more or less general, expressions shown in Eq. (2.32) and Eq. (2.33), Bernstein and Rabinowitz choose, for the sake of simplicity, the following monoenergetic distribution:

$$f^-(E_\perp, E_\parallel, J) = f^-(E_\perp, E_\parallel) = \frac{n_{i0} m_i}{2\pi} \frac{G(E_\parallel)}{\int_0^\infty dE_\parallel \frac{G(E_\parallel)}{\sqrt{2m_i E_\parallel}}} \delta(E_\perp - E_0) \tag{2.34}$$

Now, by introducing Eq. (2.34) into Eq. (2.32), after some tedious calculus the ion density can be expressed as:

$$\begin{aligned} n_i(r) = \frac{n_{i0}}{\pi} \arcsin\left(\sqrt{\frac{G(E_0}{2m_i r^2 (E_0 - e\varphi(r))}}\right) + \frac{2n_{i0}}{\pi} H\left(E_0 - e\varphi(r) - \frac{er}{2}\frac{d\varphi(r)}{dr}\right) \\ \cdot \left[\frac{\pi}{2} - \arcsin\left(\sqrt{\frac{G(E_0)}{2m_i r^2 (E_0 - e\varphi(r))}}\right)\right] \end{aligned} \tag{2.35}$$

and by taking Eq. (2.34) into Eq. (2.33), the ion flux results:

$$\vec{\Gamma}(r) = -\vec{e_r} \frac{n_{i0}\sqrt{G(E_0)}}{m_i r} \tag{2.36}$$

Once we have Eq. (2.35) and Eq. (2.36) the model of Bernstein and Rabinowitz is almost complete. From Eq. (2.36), the total ion current collected by a cylindrical probe of length L is readily obtained as:

$$\boxed{I_i = \frac{2\pi L n_{i0}\sqrt{G(E_0)}}{m_i}} \tag{2.37}$$

In their original work, Bernstein and Rabinowitz also solved Poisson's equation in order to obtain the electric potential profile along the sheath. For the sake of briefness, we are not going to cover here their model to that extent. Nevertheless, once we have an expression for the ion density given by Eq.

(2.35), the process is rather simple. Eq. (2.35) is taken into Eq. (2.20) along with the number density of electrons, assumed to be Maxwellian, and then the electric potential is obtained by numerical integration in a similar fashion than we did in the first chapter.

The model of Bernstein and Rabinowitz was extended later on by S. H. Lam [28] in 1965. Lam though that, because of the numerical nature of the solutions found by Bernstein and Rabinowitz, their results where not very practical, as they require a lot of cross-plotting in order to determine any information of interest. The starting point for Lam was almost the point where we have finished. He used the ion density obtained by Bernstein and Rabinowitz and introduced it into Poisson's equation, but instead of solving the problem numerically, he considered the case $r_p \gg \lambda_D$, and obtained analytical results for this limiting case. He also stablished a criterion for trapped ions. Finally, the most thorough orbital study was performed by James G. Laframboise, whose model we are going to review next.

2.3.3. Laframboise model

Laframboise developed his model as part of his doctoral thesis. He published it in a report at the University of Toronto Institute for Aerospace Studies [25] in 1966. Laframboise proposed a method for obtaining theoretical predictions of the current collected by an electrically conducting Langmuir probe from a fully Maxwellian plasma at rest. With his method he determined full $I-V$ characteristic curves for probe biasing potentials between $\varphi_p = -25k_BT_e$ to $\varphi_p = +25k_BT_e$, for both spherical and cylindrical geometry, and for probe radii up to $100\lambda_D$ and a complete range of ion to electron temperature ratios.

From the physic point of view, Laframboise's model is not any different than the one developed by Bernstein and Rabinowitz. Actually, it is exactly the same, Laframboise proposes to solve both the corresponding Boltzmann's equation along with Poisson's equation. Also, both models rely on the same working hypotheses, and perform the same exhaustive analysis of all the possible trajectories of particles in terms of the effective potential energy. The main difference between both models is that Laframboise though that the monoenergetical ion distribution is not always the most suitable distribution to describe ions in the plasma. For this reason he considered a fully Maxwellian plasma, *i. e.* a plasma where ions and electrons are described by a Maxwellian distribution.

$$f(E,J) = f(E) = n_0 \left(\frac{m}{2\pi k_B T}\right)^{3/2} \exp\left(-\frac{E}{k_B T}\right) \tag{2.38}$$

So, instead of a monoenergetic distribution function, Laframboise considered ions to have a distribution like the one in Eq. (2.38), and solved Eq. (2.32) and Eq. (2.33) with it. We have to notice that Eq. (2.38) must be particularised for ions or electrons as needed. Also, continuing with the notation that we have been using, we have that $E = E_\perp + E_\parallel$. The problem of using a Maxwellian distribution in Eq. (2.32) and Eq. (2.33) is that they are no longer analytically integrable. As Laframboise probed in his work, when a poly-energetic distribution is considered, the charge density at any given radius can be shown to be dependent not only on the local value of the potential at that radius, but on the value of the potential everywhere in the vicinity of the probe.

Because of the lack of an analytical expression for the ion density, the system of equations (Boltzmann and Poisson's equation) can not be reduced to a single ordinary differential equation. Instead, a nonlinear system of integral equations results, which needs to be solved numerically. The main contribution of Laframboise was to develop an iterative numerical algorithm that allowed its solution. This iterative procedure is as follows:

- An initial trial function is assumed for the net charge density.
- Poisson's equation is integrated numerically to provide the electric potential and its first two radial derivatives, as a function of the radial distance to the probe.
- Using the results from the previous step the ion and electron collected currents and charge densities are calculated. This is done by solving numerically Eq. (2.32) and Eq. (2.33).
- The resulting net charge density function is mixed with the previously used net charge density in order to provide a closer approximation to the real solution.

The steps that we have just explained here are then repeated until the approximate solutions are close enough to the real ones. The process of calculating the ion and electron charge densities from a given net density and subtracting them to give a new net density, defines a non-linear integral operator Φ, which acts on the nth net charge density approximation, $\rho_n(r)$, to give the next approximation $\rho_{n+1}(r)$. The solution of the system would be a function that satisfies the condition $\rho(r) = \Phi\rho(r)$. Nevertheless, Laframboise stated that, in general, the sequence of functions generated by the operator Φ diverges by overshooting the real solution and oscillating around it with increasing amplitude. So, he defined a mixing function $M(r)$ which had the property $0 < M(r) \leq 1$ for any r. And finally, with this function he defined a new iterative scheme as follows:

$$\rho_{n+1}(r) = M(r)\Phi\rho_n(r) + (1 - M(r))\rho_n(r) \tag{2.39}$$

If we observe Eq. (2.39), we can see that if $\rho_{n+1}(r) = \rho_n(r)$ then $\rho_n(r) = \Phi\rho_n(r)$, which is the required condition for the solution to be correct. The optimisation of the function $M(r)$ was carried out by computational experimentation.

With the described method, Laframboise obtained the data related in the first paragraph of this section. A remark that needs to be made is that, Laframboise applied his method to some conditions where trapped ion orbits may occur. Even though he acknowledge that the population of those orbits can greatly affect the prediction of the model, he believed that there are situations where the impact of such orbits could be neglected. Anyway, he stated that more theoretical work about the population of such orbits is needed in order to properly study these cases.

2.4. Radial theories of the ion current collected by a cylindrical probe

Contrary to orbital theories, radial theories are those that constrict the movement of ions to the radial dimension. So, in radial theories, the orbital motion of ions around the probe (cylindrical or spherical) is neglected. These theories were developed because there are plenty of cases where the ion temperature is much smaller than the electron temperature, *i. e.* $T_i/T_e \to 0$. Under this condition, it seems to be reasonable to neglect the motion of ions in the plasma. By considering the variables that we have been dealing with in the previous section, radial models can also be defined as, models where the angular momentum of ions, J, is assumed to be zero.

The first radial model of the ion saturation zone of the $I - V$ characteristic curve was developed in 1957 by J. E. Allen, R. L. F. Boyd and P. Reynolds [29]. This model, subsequently known as the ABR model, was developed for the case of spherical probes and later on extended in 1965 by F. F. Chen [30] for the case of cylindrical probes. Those two models considered the ion temperature to be exactly zero, but in 1996 J. I. Fernández Palop [14] developed a radial model of a cylindrical Langmuir probe considering the case of finite ion temperature. Finally in 2004 R. Morales Crespo [31] extended the model of Fernández Palop by obtaining analytical fits of the $I - V$ characteristic curves for spherical and cylindrical probes.

2.4.1. Allen, Boyd and Reynolds / Chen model (ABR model)

Although the ABR model was originally developed by Allen, Boyd and Reynolds in 1957 [29] for the case of spherical probes, since we are interested in the cylindrical geometry, we are going to center our discussion in the extension developed by Chen in 1965 [30]. Nevertheless, since the theoretical foundations are exactly the same, we are going to refer to this model as the ABR theory, even in the case of cylindrical geometry.

As we have previously introduced, the main trait of the ABR model is that the temperature of ions is considered to be zero, $T_i = 0$. Also, the previous assumption is equivalent to consider the angular momentum of ions to be zero, $J = 0$. So, in the ABR model the only conservation law that needs to be taken into account is the conservation of energy. The other equation that is considered in the model is Poisson's equation, which takes into account the electric potential distribution.

Due to the equations considered, this model is very similar to the one developed in section 1.4 for the

case of planar geometry. Nevertheless, we are going to develop the full model here in order to highlight the difference between planar and cylindrical (or spherical) geometry. In particular we are going to see that the planar case is the only one where an "external" presheath mechanism, *i. e.* ionisation, is needed in order to increase the ion current from its value at the plasma, which is zero, to the value of the current drained by the probe. Let us remember here that, when considering the planar case, in order to develop a sheath, ions needed to enter the sheath with a certain velocity, *i. e.* Bohm velocity, and ionisation allowed the ions to increase their velocity without the need of decrease their density to the same extent, as we saw in Eq. (1.17). But, if we write the continuity equation for the case of cylindrical geometry without ionisation we have that:

$$\vec{\nabla} \cdot \vec{\Gamma}_i(\vec{r}) = \frac{1}{r}\frac{\partial}{\partial r}(r\Gamma_i(r))) = \frac{\Gamma_i(r)}{r} + \frac{d\Gamma_i(r)}{dr} = 0 \Rightarrow \frac{1}{\Gamma_i(r)}\frac{d\Gamma_i(r)}{dr} = -\frac{1}{r} \Rightarrow \Gamma_i(r) \propto \frac{1}{r} \tag{2.40}$$

where $\vec{\Gamma}_i(\vec{r})$ is the ion flux at position $\vec{r}$, which obviously, because of symmetry reasons, has a radial direction and only depends on the radial distance to the axis of the probe. So, $\vec{\Gamma}_i(\vec{r}) = \Gamma_i(r)\vec{e}_r$.

As we can see in Eq. (2.40), the ion flux increases as we approach the surface of the probe simply because of the geometry of the problem. The expression we have found in Eq. (2.40) also fulfil the condition that the flux at the plasma, *i. e.* at $r \to \infty$, must be zero. If we take into account that $\Gamma_i(r) = n_i(r)v_i(r)$, the fact that the geometry allows $\Gamma_i(r)$ to increase as we decrease r, means that the ion velocity is able to increase without causing a proportional decrease in the ion density. This is precisely the effect that a presheath mechanism should have, and the role that ionisation plays in the planar case.

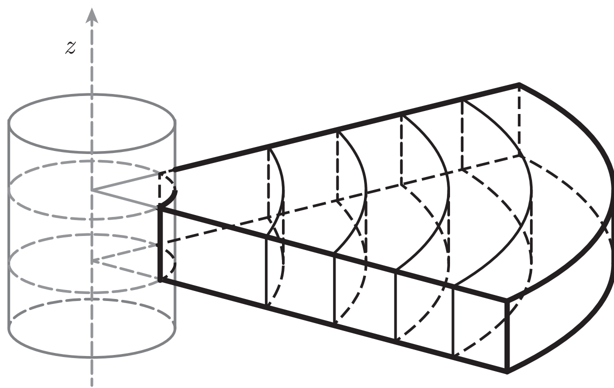


Figure 2.7: Diminishing of the volume elements as the surface of the cylindrical probe is approached.

Graphically it is easy to understand how the ion flux could be increased simply because of the geometry without the need to create more particles. In Fig. 2.7 we can see how the volume elements decrease as we approach the probe. If we imagine a certain number of ions moving inwards to the probe with a fixed radial velocity, it is clear that when the number of ions is divided by the volume element in order to obtain the ion number density this quantity is going to increase, causing the flux to increase. This same reasoning is applicable to the case of spherical geometry.

Now without further ado, once we have seen why there is no need to include an ionisation term in the continuity equation for the case of a cylindrical probe (the same would apply for a spherical one), we can start developing the ABR model. Let us start with the energy conservation law for ions. If we take the plasma as the reference for the electric potential, $\varphi_0 = 0$, and we also consider that there is no thermal motion of ions at all in the plasma, we can integrate the momentum balance equation neglecting the diffusion term (see Appendix A) in order to obtain the following energy conservation equation:

$$\frac{1}{2}m_i v_i^2(r) + e\varphi(r) = 0 \tag{2.41}$$

where v_i is the radial flow velocity of ions, the only one we are considering, and r is the radial distance to the axis of the probe.

On the other hand, continuity equation for ions can be written in the following way:

$$i = 2\pi r e\Gamma(r) = 2\pi r e n_i(r)v_i(r) \tag{2.42}$$

where i is the ion current collected by the probe per unit length. Let us notice that in Eq. (2.42) the ion current is evaluated at a position r, nevertheless, as ions are forced to approach the probe radially, the same ion current crossing the outer concentric surface shown in Fig. 2.7 must cross all of them until the surface of the probe is reached.

Now if we solve Eq. (2.42) for $v_i(r)$, take it into Eq. (2.41) and then solve Eq. (2.41) for $n_i(r)$ we

can obtain the following expression for the ion number density at r:

$$n_i(r) = \frac{i}{2\pi re}\sqrt{-\frac{m_i}{2e\varphi(r)}} \tag{2.43}$$

Once we have an expression for the ion density, we need an analogous expression for the electron density in order to solve Poisson's equation. Considering that we are studying the ion saturation zone, so that the electric potential is retarding for electrons, by using the same arguments of section 1.4 we can write the electron density as:

$$n_e(r) = n_{e0}\, \exp\left(\frac{e\varphi(r)}{k_B T_e}\right) \tag{2.44}$$

The last step that remains in order to complete the ABR model is to take Eq. (2.43) and Eq. (2.44) into Poisson's equation:

$$\begin{aligned}\nabla^2\varphi(\vec{r}) &= \frac{1}{r}\frac{d}{dr}\left(r\frac{d\varphi(r)}{dr}\right) = \frac{1}{r}\frac{d\varphi(r)}{dr} + \frac{d^2\varphi(r)}{dr^2} = -\frac{e}{\varepsilon_0}(n_i(r) - n_e(r)) \\ &= -\frac{e}{\varepsilon_0}\left[\frac{i}{2\pi re}\sqrt{-\frac{m_i}{2e\varphi(r)}} - n_{e0}\, \exp\left(\frac{e\varphi(r)}{k_B T_e}\right)\right] \Rightarrow \\ \Rightarrow \frac{1}{r}\frac{d\varphi(r)}{dr} + \frac{d^2\varphi(r)}{dr^2} &= -\frac{e}{\varepsilon_0}\left[\frac{i}{2\pi re}\sqrt{-\frac{m_i}{2e\varphi(r)}} - n_{e0}\, \exp\left(\frac{e\varphi(r)}{k_B T_e}\right)\right]\end{aligned} \tag{2.45}$$

Now all that rest is to solve Eq. (2.45) numerically, but before doing so, let us define some dimensionless variables in order to simplify Eq. (2.45):

$$R = \frac{r}{\lambda_D}\ ; \quad \psi(R) = \frac{e\varphi(r)}{k_B T_e}\ ; \quad I = \frac{i}{2\pi k_B T_e\sqrt{\frac{2\varepsilon_0 n_{e0}}{m_i}}} \tag{2.46}$$

By considering these definitions, Eq. (2.45) can be written as:

$$\frac{1}{R}\frac{d\psi(R)}{dR} + \frac{d^2\psi(R)}{dR^2} = e^{\psi(R)} - \frac{I}{R\sqrt{-\psi(R)}} \tag{2.47}$$

In order to solve Eq. (2.47) numerically, initial conditions for the dimensionless potential and its first derivative are needed, so let us figure out both of them. As we are interested in the evolution of the potential along the sheath, we are going to obtain those initial conditions from the quasineutral solution. The quasineutral solution is obtained by considering the neutrality conditions and, thus, by cancelling the right hand side of Eq. (2.47), so:

$$e^{\psi(R)} - \frac{I}{R\sqrt{-\psi(R)}} = 0 \Rightarrow R = \frac{I}{e^{\psi(R)}\sqrt{-\psi(R)}} \tag{2.48}$$

Now, if we derive Eq. (2.48) with respect to R and solve for $d\psi(R)/dR$ we get:

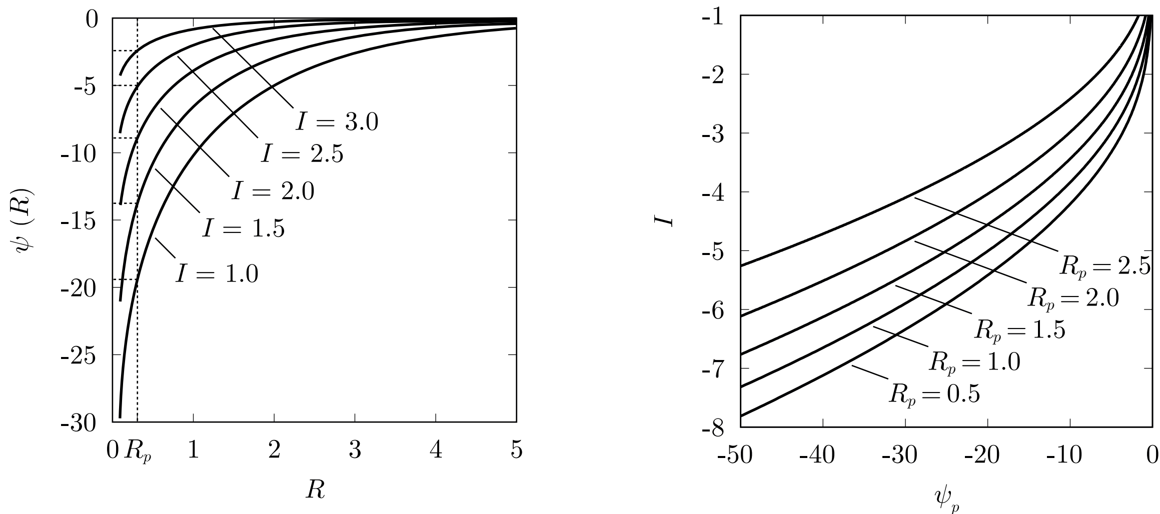
$$\frac{d\psi(R)}{dR} = \frac{e^{\psi(R)}\sqrt{-\psi(R)}}{-I\left[1 + \dfrac{1}{2\psi(R)}\right]} \tag{2.49}$$

Eq. (2.48) and Eq. (2.49) give us the initial conditions that we need for the electric potential as well as its first derivative. Because of the impossibility of explicitly solving Eq. (2.48) for the potential, instead of taking a fixed initial position and obtaining the initial condition for ψ and $d\psi/dR$, the following process is followed:

- Because $\psi \to 0$ and $d\psi/dR \to 0$ as $R \to \infty$, a small value is fixed as initial condition for the potential, ψ_s.
- With Eq. (2.48) and ψ_s, the position of the initial condition, R_s, is located.

- By taking the values R_s and ψ_s into Eq. (2.49) the initial condition for the derivative of the potential, $(d\psi/dR)_s$, is obtained.

We have to notice, that Eq. (2.47) is solved for a fixed value of the dimensionless current I. Once we fix the I value and follow the previous steps, in order to find the proper initial conditions, the evolution of the potential is easily obtained through numerical integration (RK4). In Fig. 2.8a different solutions of Eq. (2.47) for different values of I are shown. In Fig. 2.8b the $I-V$ characteristic curves for different probe radii can be seen. Those $I-V$ characteristic curves are found by cross-plotting of the potential curves. The process of cross-plotting is shown graphically in Fig. 2.8a. Once we fix a probe radius, R_p, different pairs of (ψ_p, I) values can be found, providing I as a function of ψ_p. We have to notice that in Fig. 2.8b the values of the current are assumed to be negative for the sake of consistence with Fig. 1.12, even though in the equations used in this section $I > 0$ by definition.



(a) Electric potential across the sheath.

(b) $I-V$ characteristic curves found by cross-plotting.

Figure 2.8: Solutions of the ABR model for a cylindrical probe obtained through numerical integration (RK4) of Eq. (2.47).

As we stated, the ABR model that we have just presented here, is developed for the case of completely cold ions, $T_i = 0$. But, it might worth to ask the question wether it would be reasonable to consider a radial model, that is neglecting the orbital motion of ions, when ions have a finite temperature. That question was addressed by Fernández Palop *et. al.* [14] and his model is the one we are going to review next.

2.4.2. Fernández Palop model

The most complete theoretical study of the ion current collected by a probe is the model developed by Laframboise [25], which is based on the framework stablished by Bernstein and Rabinowitz [24], both of which have been reviewed in previous sections. However, after the development of the ABR model, several experimental results [32–34] found that the predictions of the ABR theory were closer to the measured ion currents than any orbital model. For that reason, in 1996 J. I. Fernández Palop *et. al.* [14] developed an extension of the ABR model where a finite ion temperature was considered and the orbital motion of ions around the probe was diminished.

The model starts by considering Poisson's equation in cylindrical coordinates, which, taking into account the symmetry of the problem, can be written as:

$$\frac{1}{r}\frac{d}{dr}\left(r\frac{d\varphi(r)}{dr}\right) = -\frac{e}{\varepsilon_0}[n_i(r) - n_e(r)] \tag{2.50}$$

Electrons are assumed to be in thermal equilibrium with the electric field, and the biasing potential of the probe is assumed to be negative enough so that all the electrons approaching the probe are repelled

back into the plasma. Under this conditions, the electron density will be:

$$n_e(r) = n_{e0} \exp\left(\frac{e\varphi(r)}{k_B T_e}\right) \tag{2.51}$$

Now, in order to obtain an expression for the ion density, we consider the continuity equation and conservation of energy, like we did in the ABR model. The main difference is that, the momentum balance equation that we are going to use for describing an steady flow of ions moving radially towards a cylindrical probe immersed in a collisionless plasma is, has a diffusion term in it:

$$n_i(r)v_i(r)\frac{dv_i(r)}{dr} + \frac{1}{m_i}\frac{dp_i(r)}{dr} + \frac{e}{m_i}n_i(r)\frac{d\varphi(r)}{dr} = 0 \tag{2.52}$$

where $p_i(r)$ is the partial pressure of ions. As we are going to consider ions at a certain finite temperature, we are not neglecting the diffusion term in Eq. (2.52) (see Appendix A for a discussion about this issue). In order to obtain an expression for the ion partial pressure, we are going to consider an adiabatic flow for ions, since it has been shown by Riemann [35] and Zawaideh *et. al.* [36] that the fluid approximation in the sheath is not consistent otherwise. Therefore, the ion pressure is related to the ion density as:

$$p_i(r) \propto n_i^{\gamma}(r) \tag{2.53}$$

which, by taking into account the state equation for the ion fluid as well as the neutrality condition in the plasma, can be written as:

$$p_i(r) = k_B T_i n_{e0}^{1-\gamma} n_i^{\gamma}(r) \tag{2.54}$$

We have to notice that $\gamma = 3, 2, 5/3$ for one, two or three degrees of freedom respectively. In our case, we are considering an infinite cylindrical probe, so, we have a compression of the ion fluid in the two dimensions perpendicular to the axis of the probe. As it is shown in Fig. 2.7, no compression is suffered along the axial dimension. Accordingly, we could think that the value $\gamma = 2$ should be used. However, when the probe radius is large enough compared to the Debye length, the problem becomes almost monodimensional and $\gamma = 3$ should be used. We are going to present the model equations for two cases: the general bidimensional case when $r_p \lesssim \lambda_D$ and the monodimensional case when $r_p \gg \lambda_D$, r_p being the probe radius.

Now, if we take Eq. (2.54) into Eq. (2.52), after integration, the following energy balance equation is obtained:

$$\frac{1}{2}m_i v_i^2(r) + k_B T_i \frac{\gamma}{\gamma - 1}\left(\frac{n_i(r)}{n_{e0}}\right)^{\gamma-1} + e\varphi(r) = k_B T_i \frac{\gamma}{\gamma - 1} \tag{2.55}$$

The last equation to consider is the continuity equation for ions, which is exactly the same that the one used in the ABR model, given by Eq. (2.42). By taking it into Eq. (2.55), yields:

$$\frac{1}{2}\frac{m_i i^2}{(e2\pi r n_i(r))^2} + k_B T_i \frac{\gamma}{\gamma - 1}\left(\frac{n_i(r)}{n_{e0}}\right)^{\gamma-1} + e\varphi(r) = k_B T_i \frac{\gamma}{\gamma - 1} \tag{2.56}$$

Now, we would have to solve Eq. (2.56) for $n_i(r)$, introduce its value into Poisson's equation and solve it numerically. The problem is that Eq. (2.56) is not straightforward to solve, so in order to clarify the problem we are going to use the usual dimensionless variables defined as:

$$R = \frac{r}{\lambda_D} \;;\quad \psi(R) = \frac{e\varphi(r)}{k_B T_e} \;;\quad \beta = \frac{T_i}{T_e} \;;\quad N_i(R) = \frac{n_i(r)}{n_{e0}} \;;\quad I = \frac{ie}{2\pi\varepsilon_0}\left(\frac{m_i}{2k_B^3 T_e^3}\right)^{1/2} \tag{2.57}$$

With the use of the previous dimensionless variables, Eq. (2.56) can be expressed as follows:

$$\frac{I^2}{R^2} + \frac{\gamma}{\gamma - 1}\beta N_i^{\gamma+1}(R) + \psi(R)N_i^2(R) = \frac{\gamma}{\gamma - 1}\beta N_i^2(R) \tag{2.58}$$

Let us notice that, when it comes to solve Eq. (2.58) for the ion density, depending on the value of γ considered, the equation becomes a quartic ($\gamma = 3$) or a cubir ($\gamma = 2$). Considering either case there is,

in general, a multiplicity of solutions for the ion density. The meaningful solution from a physical point of view should recover the ABR theory in the limit $\beta \to 0$, that is:

$$\lim_{\beta \to 0} N_i(R) \sim \frac{I}{R\sqrt{-\psi(R)}} \tag{2.59}$$

There is only one solution among those of Eq. (2.58) that recovers the right behaviour of Eq. (2.59) (further details can be found in the original work of Fernández Palop *et. al.* [14]). For the case of $\gamma = 2$ we have that:

$$N_i(R) = \left\{ \sqrt{\frac{2\beta - \psi(R)}{S}} \left[\cos\left(\frac{\theta}{3}\right) - \frac{\sqrt{3}}{3} \sin\left(\frac{\theta}{3}\right) \right] \right\}^{-1} \tag{2.60}$$

and for $\gamma = 3$:

$$N_i(R) = \left[\frac{3\beta/2 - \psi(R) + \sqrt{(3\beta/2 - \psi(R))^2 - 6\beta S}}{2S} \right]^{-1/2} \tag{2.61}$$

where:

$$S = \left(\frac{I}{R}\right)^2 \quad ; \quad \theta = \arcsin\left(\frac{3\sqrt{3}\beta S}{(2\beta - \psi(R))^2} \sqrt{\frac{2\beta - \psi(R)}{S}} \right) \tag{2.62}$$

Now that we have expressions for both, the ion and the electron density (in dimensionless units $N_e(R) = \exp(\psi(R))$), all that remains is to solve Poisson's equation, which can be written in terms of the dimensionless variables as:

$$\frac{1}{R}\frac{d\psi(R)}{dR} + \frac{d^2\psi(R)}{dR^2} = e^{\psi(R)} - N_i(R) \tag{2.63}$$

Eq. (2.63) can now be solved numerically for the case $\gamma = 2$ by using Eq. (2.60) and for the case $\gamma = 3$ by using Eq. (2.61). Nevertheless, before solving Eq. (2.63), we need to find proper initial conditions, as we did when solving the ABR model. Let us start with the general case of bidimensional adiabatic flow.

Initial conditions for the case of bidimensional adiabatic flow ($\gamma = 2$, $r_p \lesssim \lambda_D$)

The initial conditions for solving Eq. (2.63) must be taken far away from the probe. So, in order to find them, we consider the quasineutral condition that must be hold in the plasma the same way we did before, that is by considering the right hand side of Eq. (2.63) to be zero. If we consider the expression for the ion density given by Eq. (2.60), the quasineutral condition can be written as:

$$\lim_{R \to \infty} e^{-\psi(R)} \sim \sqrt{\frac{2\beta - \psi(R)}{S}} \left[\cos\left(\frac{\theta}{3}\right) - \frac{\sqrt{3}}{3} \sin\left(\frac{\theta}{3}\right) \right] \tag{2.64}$$

where S and θ are defined in Eq. (2.62).

Now, let us analyse the quasineutral solution given by Eq. (2.64). The first thing we have to notice is that, as the argument of the arcsin function in Eq. (2.62) is always positive, its value must range between zero and one, so:

$$\frac{3\sqrt{3}\beta S}{(2\beta - \psi(R))^2} \sqrt{\frac{2\beta - \psi(R)}{S}} \leq 1 \Rightarrow \frac{3\beta}{2\beta - \psi(R)} \leq \frac{1}{\sqrt{3}} \sqrt{\frac{2\beta - \psi(R)}{S}} \tag{2.65}$$

Also, precisely because the argument of the arcsin function in Eq. (2.62) ranges between 0 and 1, we have that $\theta \in [0, \pi/2]$. Once we know that, it can be easily probed that the term in brackets in Eq. (2.64) ranges between 1 and $\sqrt{3}/3$, having the latter when the argument of the arcsin function is 1. So, Eq. (2.64) must fulfil the condition:

$$e^{-\psi(R)} \geq \frac{1}{\sqrt{3}} \sqrt{\frac{2\beta - \psi(R)}{S}} \tag{2.66}$$

Finally, in order for Eq. (2.64) to have a solution, we have that, from Eq. (2.65) and Eq. (2.66), the electric potential must fulfil the condition:

$$e^{-\psi(R)} \geq \frac{3\beta}{2\beta - \psi(R)} \tag{2.67}$$

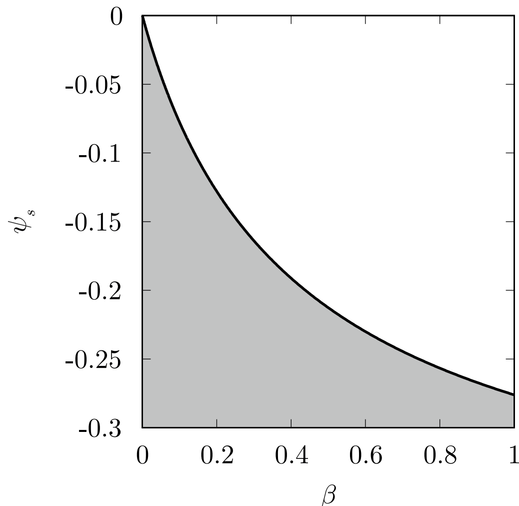


Figure 2.9: Plot of Eq. (2.67). The greyed area correspond to the inequality while the solid line corresponds to the equal sing.

We shall notice that the previous equation establishes a minimum value for the initial condition of the potential, ψ_s. Contrary to what happened in the ABR theory, in this model we can no longer consider the initial conditions to be as close to the plasma as we want, that is $\psi_s \rightarrow 0$. Instead, Eq. (2.67) has to be taken into account. In Fig. 2.9 Eq. (2.67) is plotted. The greyed area represent the values of the potential that fulfil the inequality given by Eq. (2.67), while the solid line corresponds to Eq. (2.67) when the equal sign is considered. The solid line shows the minimum value of the potential that we can use as initial condition when solving Eq. (2.63) for each β value. In Fig. 2.9 it can also be seen that, when $\beta = 0$, ψ_s can be as small as we want, which is the right behaviour corresponding to the ABR model. We have to notice that, when Poisson's equation is solved numerically, the initial condition for the electric potential has to be slightly larger than the minimum value predicted by Eq. (2.67).

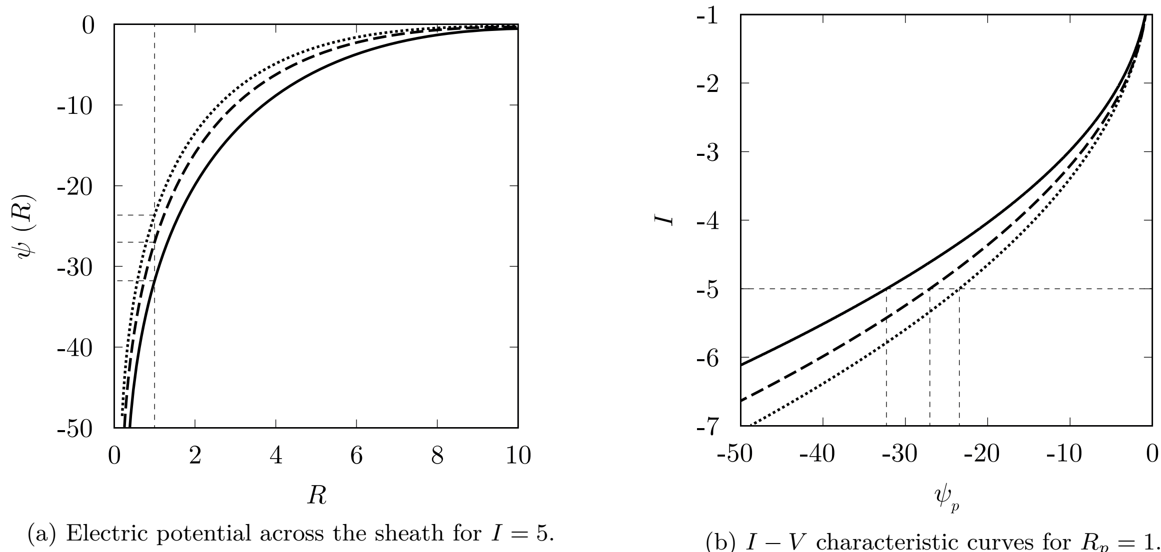
Once we know the value of the initial condition for the electric potential, if we take the asymptotic behaviour of $N_i(R)$ into Eq. (2.58) for $\gamma = 2$ we obtain the position for the initial condition as:

$$R_s = I\left[(2\beta - \psi_s)e^{2\psi_s} - 2\beta e^{3\psi_s}\right]^{-1/2} \tag{2.68}$$

and by differentiating Eq. (2.68) the initial condition for the derivative of the potential can also be found:

$$\left(\frac{d\psi(R)}{dR}\right)_s = \frac{2I^2}{R_s^3}\left[6\beta e^{3\psi_s} - (4\beta - 1 - 2\psi_s)e^{2\psi_s}\right]^{-1} \tag{2.69}$$

Finally, Poisson's equation can be solved by numerical integration in the same fashion as we did in the ABR model. In Fig. 2.10 some solutions can be seen showing the dependence of the results with the ion temperature. As we said in the ABR model, in Fig. 2.10b the values of the current are shown as negatives because it is due to ions, however in the equations of this section I is strictly positive.



(a) Electric potential across the sheath for $I = 5$.

(b) $I - V$ characteristic curves for $R_p = 1$.

Figure 2.10: Solutions of the model for bidimensional adiabatic flow ($\gamma = 2$, $r_p \lesssim \lambda_D$) obtained by numerical integration (RK4) of Eq. (2.63) and cross plotting. Solid lines $\beta = 0$, dashed lines $\beta = 0.2$ and dotted lines $\beta = 0.4$.

Initial conditions for the case of monodimensional adiabatic flow ($\gamma = 3$, $r_p \gg \lambda_D$)

The process of finding proper initial conditions for solving Poisson's equation in this case is very similar to the one developed in the previous case. If we consider the expression for the ion density given by Eq. (2.61), the quasineutral condition can be written as:

$$\lim_{R\to\infty} e^{-\psi(R)} = \left[\frac{3\beta/2 - \psi(R) + \sqrt{(3\beta/2 - \psi(R))^2 - 6\beta S}}{2S}\right]^{1/2} \tag{2.70}$$

where S and θ where defined in Eq. (2.62).

Now, if we analyse the quasineutral solution given by Eq. (2.70), we notice that the argument of the square root inside the brackets has to be positive, that is:

$$\left(\frac{3\beta}{2} - \psi(R)\right)^2 - 6\beta S \geq 0 \Rightarrow \left(\frac{3\beta}{2} - \psi(R)\right)^2 \geq 6\beta S \tag{2.71}$$

Finally, if we take the condition given by Eq. (2.71) into the quasineutral solution given by Eq. (2.70) we find the condition that the electric potential must fulfil in order for Eq. (2.70) to have a solution:

$$e^{-2\psi(R)} \geq \frac{3\beta}{3\beta/2 - \psi(R)} \tag{2.72}$$

We shall notice that, as in the bidimensional case, Eq. (2.72) establishes a minimum value for the initial condition of the potential, ψ_s, and the same behaviour found in the previous case is found for the monodimensional one. In Fig. 2.11 Eq. (2.72) is plotted. The greyed area represent the values of the potential that fulfils the inequality given by Eq. (2.72), while the solid line corresponds to Eq. (2.72) when the equal sing is considered. The solid line shows the minimum value of the potential that we can use as initial condition when solving Eq. (2.63) for each β value. In Fig. 2.11 it can also be seen that, when $\beta = 0$, ψ_s can be as small as we want, which is, again, the right behaviour corresponding to the ABR model. Also, as in the previous case, the initial condition for the numerical resolution of Poisson's equation must be taken slightly larger than the minimum value predicted by Eq. (2.72).

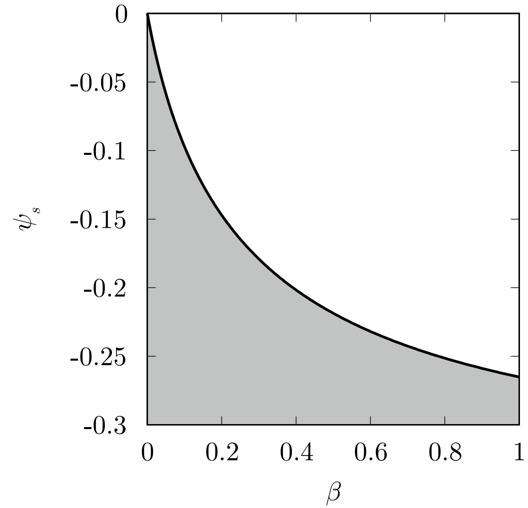


Figure 2.11: Plot of Eq. (2.72). The greyed area correspond to the inequality while the solid line corresponds to the equal sing.

Once we know the value of the initial condition for the electric potential, if we take the asymptotic behaviour of $N_i(R)$ into Eq. (2.58) for $\gamma = 3$ we obtain the position for the initial condition as:

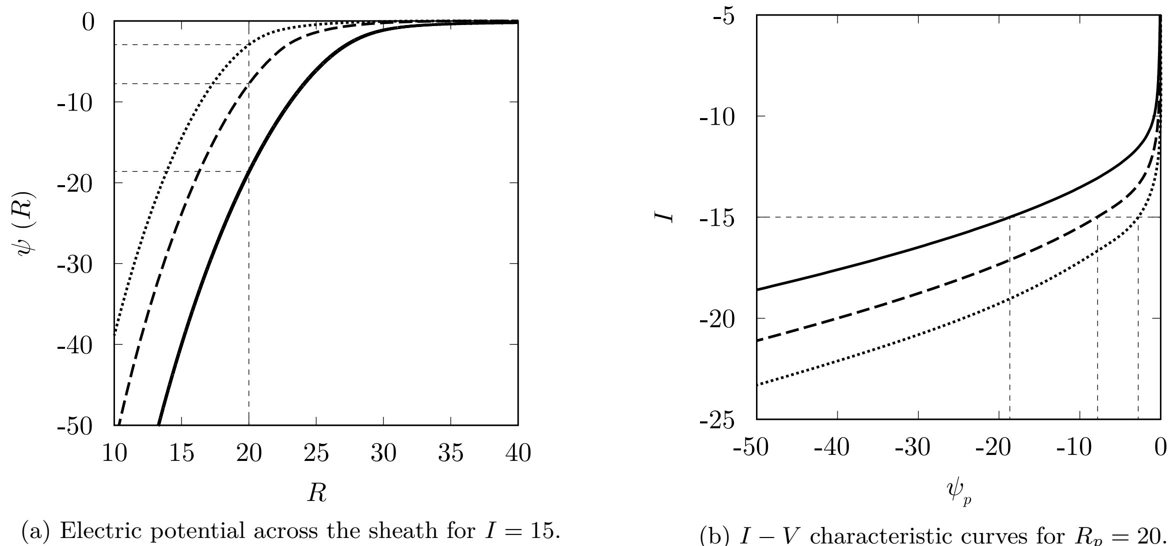
$$R_s = I\left[\left(\frac{3\beta}{2} - \psi_s\right)e^{2\psi_s} - \frac{3\beta}{2}e^{4\psi_s}\right]^{-1/2} \tag{2.73}$$

and by differentiating Eq. (2.73) the initial condition for the derivative of the potential can also be found as:

$$\left(\frac{d\psi(R)}{dR}\right)_s = \frac{2I^2}{R_s^3}\left[6\beta e^{4\psi_s} - (3\beta - 1 - 2\psi_s)e^{2\psi_s}\right]^{-1} \tag{2.74}$$

Finally, Poisson's equation can also be solved by numerical integration for the case of monodimensional adiabatic flow. In Fig. 2.12 some solutions can be seen showing the dependence of the results with the ion temperature. Again, in Fig. 2.12b the values of the current are shown as negatives because it is due to the ions, however in the equation of this section I is strictly positive.

The model developed by Fernández Palop, shares a difficulty with most of the theories that we have reviewed in this chapter. As Lam stated, the numerical nature of the solutions found, as well as the need



(a) Electric potential across the sheath for $I = 15$.

(b) $I - V$ characteristic curves for $R_p = 20$.

Figure 2.12: Solutions of the model for monodimensional adiabatic flow ($\gamma = 3$, $r_p \gg \lambda_D$) obtained by numerical integration (RK4) of Eq. (2.63) and cross plotting. Solid lines $\beta = 0$, dashed lines $\beta = 0.2$ and dotted lines $\beta = 0.4$.

of cross plotting in order to obtain the $I - V$ characteristic curves, makes the use of such a model not particularly practical when it comes to diagnose a plasma. Instead, an analytical expression would be desirable for diagnostic purposes, *e. g.* the one given by Eq. (2.18) for the OML theory. It is precisely this problem what Morales Crespo's extension to this model deal with.

2.4.3. Morales Crespo model

This is the last model that we are going to review. It was developed in 2004 by R. Morales Crespo *et al.* [31] as an extension to Fernández Palop's model. The model was developed for the cases of cylindrical and spherical geometries, nevertheless, we are going to restrict our review to the cylindrical case as we have been doing in the other sections of the chapter.

Although the physics behind this model is exactly the same than the previous one, the main aim of this theory was to find analytical expressions that fit the results obtained by Fernández Palop *et al.*, so that it would be easier to diagnose a plasma. In particular, Morales Crespo found analytical expressions that approximate the electric potential and $I - V$ characteristic curves shown in Fig. 2.10. As the model relies on the same physical statements as Fernández Palop's model, the equation that needs to be solved is Poisson's equation, as given by Eq. (2.63), where the ion density is obtained by solving Eq. (2.58). It has to be noticed that Morales Crespo solves the more general case of $r_p \lesssim \lambda_D$, where the adiabatic flow of ions towards the probe is bidimensional, *i. e.* $\gamma = 2$. The initial condition for the Poisson's equation is, as usual, obtained from the quasineutral solution.

The basic idea that this model relies on is that, despite the fact that the ion thermal energy finite, it is usually small when compared to the electron one. So, under this assumption, Eq. (2.63) can be solved by using a perturbative method. Let us express an expansion of both, the electric potential, and the ion number density in the following form:

$$\psi(R, \beta) = \psi_0(R) + \psi_1(R)\beta + \psi_2(R)\beta^2 + \cdots \tag{2.75a}$$

$$N(R, \beta) = N_0(R) + N_1(R)\beta + N_2(R)\beta^2 + \cdots \tag{2.75b}$$

If we take Eqs. (2.75) into Eq. (2.58), Eq. (2.63) and the quasineutral solution, a hierarchy of sets of

equations is obtained. Those sets of equations can be written in general as:

$$N_i\left[R,\psi_i(R)\right] = N_i\left[R,\psi_0(R),\psi_1(R),\ldots,\psi_{i-1}(R),\psi_i(R)\right] \tag{2.76a}$$

$$\frac{1}{R}\frac{d\psi_i(R)}{dR} + \frac{d^2\psi_i(R)}{dR^2} = F_i\left[R,\psi_0(R),\psi_1(R),\ldots,\psi_{i-1}(R),\psi_i(R)\right] - N_i\left[R,\psi_i(R)\right] \tag{2.76b}$$

$$\lim_{R\to\infty} N_i\left[R,\psi_0(R),\psi_1(R),\ldots,\psi_{i-1}(R),\psi_i(R)\right] = F_i\left[R,\psi_0(R),\psi_1(R),\ldots,\psi_{i-1}(R),\psi_i(R)\right] \tag{2.76c}$$

Eq. (2.76a) determines the ith term of the positive ion density, N_i, as a function of R, $\psi_0(R)$, $\psi_1(R)$, ..., $\psi_{i-1}(R)$ and $\psi_i(R)$. Eq. (2.76b) corresponds to the ith term of Poisson's equation series, where $F_i[\cdots]$ is the ith term of the electron density series, $\exp(\psi(R))$. And Eq. (2.76c) represent the initial condition for the ith term of Poisson's equation. More precisely, Eq. (2.76c) represent the initial condition for the electric potential, nevertheless, the initial condition for the electric field is readily found by differentiation. If we perform the calculations, the first three sets of equations can be expressed as follows:

- Zero order term:

$$\left(\frac{I}{R}\right)^2 + \psi_0 N_0^2 = 0 \tag{2.77a}$$

$$\frac{1}{R}\frac{d\psi_0}{dR} + \frac{d^2\psi_0}{dR^2} = \exp(\psi_0) - N_0 \tag{2.77b}$$

$$\lim_{R\to\infty} N_0 = \exp(\psi_0) \tag{2.77c}$$

- First order term:

$$\psi_1 N_0^2 + 2N_1N_0\psi_0 + \frac{\gamma}{\gamma-1}\left(N_0^{\gamma-1} - N_0^2\right) = 0 \tag{2.78a}$$

$$\frac{1}{R}\frac{d\psi_1}{dR} + \frac{d^2\psi_1}{dR^2} = \psi_1\exp(\psi_0) - N_1 \tag{2.78b}$$

$$\lim_{R\to\infty} N_1 = \psi_1\exp(\psi_0) \tag{2.78c}$$

- Second order term:

$$\psi_2 N_0^2 + 2\psi_0 N_1 N_0 + \psi_0 N_1^2 + 2\psi_0 N_0 N_2 + \frac{\gamma}{\gamma-1}N_1\left(N_0^{\gamma} - 2N_0\right) = 0 \tag{2.79a}$$

$$\frac{1}{R}\frac{d\psi_2}{dR} + \frac{d^2\psi_2}{dR^2} = \left(\frac{\psi_1^2}{2} + \psi_2\right)\exp(\psi_0) - N_2 \tag{2.79b}$$

$$\lim_{R\to\infty} N_2 = \left(\frac{\psi_1^2}{2} + \psi_2\right)\exp(\psi_0) \tag{2.79c}$$

Eqs. (2.77)–(2.79) represent the hierarchy of sets of equations that we said before until the second order. We have to notice that, for the sake of clarity, in the previous equations we have omitted the implicit dependencies of ψ_i and N_i. Also, as it seems reasonable, the zero order term given by Eqs. (2.77) is equivalent to the ABR model, that we reviewed at the beginning of this section. Actually, if we take Eq. (2.77a) into Eq. (2.77b) we get Eq. (2.47).

In his original work, Morales Crespo solved Eqs. (2.77)–(2.79) and stated that the second order contribution could be diminished, so instead of Eqs. (2.75) he only considered the linear term, neglecting higher orders contributions.

$$\psi(R,\beta) \simeq \psi_0(R) + \psi_1(R)\beta \tag{2.80a}$$

$$I(R_p,\psi_p,\beta) \simeq I_0(R_p,\psi_p) + I_1(R_p,\psi_p)\beta \tag{2.80b}$$

Simply by observing Eqs. (2.77)–(2.79) it can be realised that those equations must be solved numerically. In fact, even the zero order term was integrated numerically by us when we reviewed the ABR

theory. So, Morales Crespo performed an analytical fit to Eq. (2.80b) by using the following expressions:

$$I_0(R_p,\psi_p) = \frac{1}{R_p}\sum_{i=0}^{1} c_i(R_p)\,(-\psi_p)^{i/2} \tag{2.81a}$$

$$I_1(R_p,\psi_p) = \frac{1}{R_p}\sum_{i=0}^{1} d_i(R_p)\,(-\psi_p)^{i/2} \tag{2.81b}$$

where $c_i(R_p)$ and $d_i(R_p)$ are the coefficients of the corresponding analytical fit, which depend on R_p and, in turn, are also fitted as:

$$c_i(R_p) = \sum_{j=0}^{2} c_{ij} R_p^j \tag{2.82a}$$

$$d_i(R_p) = \sum_{j=0}^{2} d_{ij} R_p^j \tag{2.82b}$$

being c_{ij} and d_{ij} the coefficients of this second fit.

Finally, let us notice that Eq. (2.80b) can be written for an arbitrary position, R, and potential, ψ. So, if we take Eqs. (2.81) into Eq. (2.80b), expand it and identify terms with Eq. (2.80a) we can obtain that:

$$\psi_0(R) = -\left[\frac{IR - c_0(R)}{c_1(R)}\right]^2 \tag{2.83a}$$

$$\psi_1(R) = 2\frac{\sqrt{-\psi_0(R)}}{c_1(R)}\left[d_0(R) + d_1(R)\sqrt{-\psi_0(R)}\right] \tag{2.83b}$$

where $c_i(R)$ and $d_i(R)$ are given by the same definitions appearing in Eqs. (2.82).

Coefficients		R_p^0	R_p^1	R_p^2
$(-\psi_p)^0$	1	$c_{00} = -3.302 \times 10^{-1}$	$c_{01} = 1.762 \times 10^{-1}$	$c_{02} = 4.601 \times 10^{-1}$
	β	$d_{00} = 9.081 \times 10^{-1}$	$d_{01} = -9.300 \times 10^{-1}$	$d_{02} = 5.080 \times 10^{-1}$
$(-\psi_p)^1$	1	$c_{10} = -2.375 \times 10^{-1}$	$c_{11} = 9.930 \times 10^{-1}$	$c_{12} = 2.228 \times 10^{-2}$
	β	$d_{10} = -2.801 \times 10^{-1}$	$d_{11} = 6.232 \times 10^{-1}$	$d_{12} = -4.232 \times 10^{-3}$

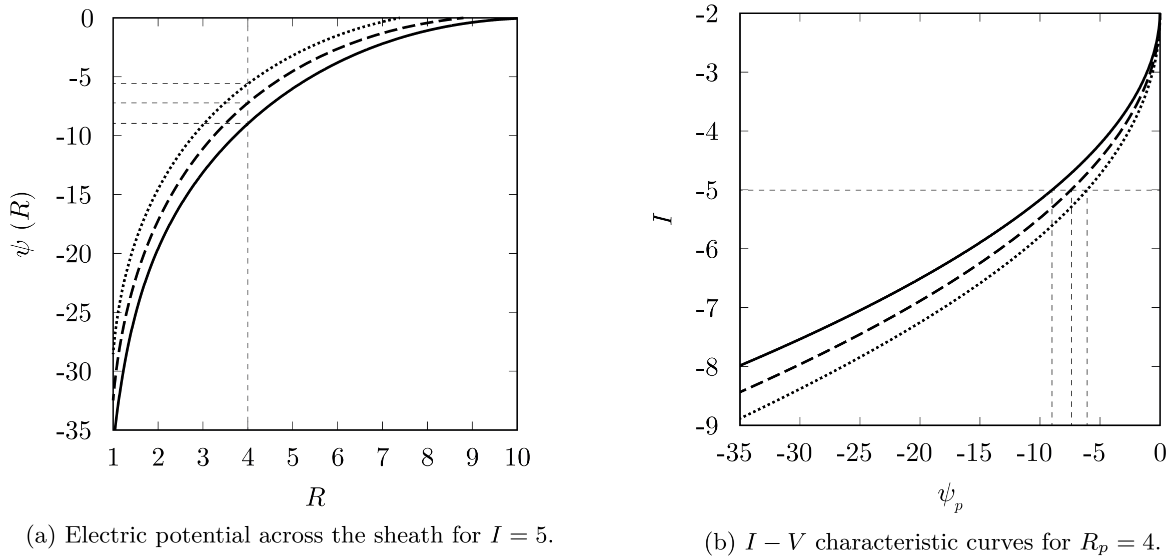
(a) Values for $R_p \in [1, 10]$

Coefficients		R_p^0	R_p^1	R_p^2
$(-\psi_p)^0$	1	$c_{00} = -6.796$	$c_{01} = 1.162$	$c_{02} = 4.194 \times 10^{-1}$
	β	$d_{00} = 29.95$	$d_{01} = -5.397$	$d_{02} = 6.728 \times 10^{-1}$
$(-\psi_p)^1$	1	$c_{10} = -2.263$	$c_{11} = 1.343$	$c_{12} = 5.623 \times 10^{-3}$
	β	$d_{10} = -4.832$	$d_{11} = 1.365$	$d_{12} = -3.441 \times 10^{-2}$

(b) Values for $R_p \in [10, 50]$

Table 2.1: Fitting coefficients for the $I - V$ characteristic curves for a cylindrical probe.

The numerical values of the coefficients found by Morales Crespo are shown in Table 2.1a for dimensionless probe radii ranging from 1 to 10, and in Table 2.1b for dimensionless probe radii ranging from 10 to 50. As it might be expected, the best behaviour of the fitting is found for R_p values centred in the



(a) Electric potential across the sheath for $I = 5$.

(b) $I - V$ characteristic curves for $R_p = 4$.

Figure 2.13: Reconstruction of the potential distribution and $I-V$ characteristic as given by Eqs. (2.80) and the coefficients in Table 2.1a. Solid lines $\beta = 0$, dashed lines $\beta = 0.1$ and dotted lines $\beta = 0.2$.

previously mentioned ranges. Also, the value of the ion temperature must be low enough when compared to the electron one, so that β is small enough for the perturbative method to make any sense.

In figure Fig. 2.13 we can see some potential profiles and $I-V$ characteristic curves obtained with the fitting coefficients found by Morales Crespo. We have to notice that the graph in Fig. 2.13a is restricted to $R \in [1, 10]$, as this is the range where the coefficients in Table 2.1a are valid.

2.5. Comparison between orbital and radial theories

In the previous sections we have reviewed the most remarkable theories that predict the ion current collected by a cylindrical Langmuir probe in the ion saturation zone. Those theoretical models can be classified as orbital or radial theories. From now on, we are going to restrict our discussion to the most representative theories of either category: the OML theory by Mott-Smith and Langmuir [7] and the ABR theory by Allen, Boyd and Reynolds / Chen [29, 30]. Obviously, with such a profound difference between the orbital and the radial approach, the results obtained when diagnosing with the OML or the ABR model are really different. For this reason, it is of particular importance to get to know the way ions really behave in order to use the proper theory. However this is not always an easy task, and experimentalists have found results in agreement with the OML [37, 38], the ABR [39, 40] or somewhere in between them [41].

One of the main problems of using the ion saturation zone of the $I-V$ characteristic in cylindrical Langmuir probes is that it is not known, *a priori*, which theoretical model must be used to perform the diagnosis. Sometimes, the use of a particular model is taken from granted as long as some conditions are met, even though the model can not be fully validated. Several authors [30, 42–45, 31, 46] have proposed different criteria to determine whether one theory or another is applicable to a particular case. However, no matter which theory is used, usually different values for the ion and electron densities are found, even though we know that $n_{e0} \simeq n_{i0}$ in order to fulfil the quasineutral condition at the plasma. This discrepancy between the densities is usually attributed to the effect of secondary emission, presence of negative ions, collisions inside the sheath, non-maxwelliam thermal distributions, so on and so forth. However, ions are usually not perfectly described by either the OML, ABR or any other model.

It has been recently found, by Díaz-Cabrera *et al.* [47], that ions experiment a transition between the radial and orbital behaviour in a Helium plasma. The transition takes place as the ratio of ion to electron temperature, $\beta = T_i/T_e$, is increased. This transition suggests that the OML and ABR theories are limiting cases of a more general model that describes the behaviour of ions around the probe. It is precisely the study of this problem what represents one of the main aims of this work.

As we have said a few lines above, there are several criteria that allow us to know if one particular model is applicable to a certain case. However, we are not only interested in knowing which theoretical framework better describes the behaviour of ions, but also in highlighting the difference between the different models themselves, in order to shed light on the pursuit of a general model that describes the behaviour of ions not only under certain conditions. For this task, the Sonin-plot representation, that we are going to showcase next, comes in handy.

2.5.1. Sonin-plot

In 1966 Ain A. Sonin published an article [48] dealing with the use of cylindrical Langmuir probes in flowing Argon plasmas. There, he studied the dependence of the ion current collected on the probe radius and the flow velocity of the plasma. In order to study the dependence with the probe radius, Sonin developed a representation of the ion current that turned out to be of great interest. This representation was latter known as Sonin-plot after him.

As we have stated, the Sonin-plot is a representation that relates the ion current collected by the probe with the probe radius. In particular, the dimensionless probe radius, $R_p = r_p/\lambda_D$, is the parameter that Sonin used in his study. To be more precise, the Sonin-plot represents the following dimensionless ion current:

$$y_{\text{sonin}}(R_p, \psi_p, \beta) = I'(R_p, \psi_p, \beta) = \frac{i}{e r_p n_{e0}} \sqrt{\frac{m_i}{2\pi k_B T_e}} \tag{2.84}$$

versus the same dimensionless current multiplied by the squared dimensionless probe radius:

$$x_{\text{sonin}}(R_p, \psi_p, \beta) = I'(R_p, \psi_p, \beta) R_p^2 = \frac{i e r_p}{\varepsilon_0} \sqrt{\frac{m_i}{2\pi k_B^3 T_e^3}} \tag{2.85}$$

where all the parameters have been previously defined along the text. We have to notice that both, the abscise as well as the ordinate of the Sonin-plot, are dimensionless quantities that depend on the same three parameters: R_p, ψ_p and β. Usually, two of these parameters are fixed, so a parametric curve is obtained. The parameters that are usually fixed are the biasing potential of the probe, ψ_p, which is to be taken in the ion saturation zone (see the end of section 2.2) and the ratio of the ion to electron temperature, which in the plasmas we are considering is usually small, $\beta < 1$.

One of the reasons why the Sonin-plot results a useful tool is because it can be used for diagnosing purposes. As J. Ballesteros *et al.* [49] proposed, the fact that the abscise of the Sonin-plot, x_{sonin}, does not depend on the electron density, n_{e0}, can be used to obtain its value with a simple cross plotting technique. However, we are interested in the Sonin-plot because its ability to highlight the differences between the predictions of different models, in particular between the radial and the orbital ones. For this reason we are going to represent the Sonin-plot corresponding to the OML and the ABR theory, which can be easily obtained from the corresponding $I-V$ characteristic.

Let us start with the OML theory. We have to remember here that the OML model could be resolved analytically, so an expression for the ion current can be obtained. Eq. (2.18) gives us the total current collected by a probe of length L. So the ion current per unit length corresponding to the OML theory is given by:

$$i_{\text{OML}} = 4 e r_p n_{i0} \sqrt{\frac{k_B T_i}{2 m_i}} \sqrt{1 - \frac{e\varphi_p}{k_B T_i}} \tag{2.86}$$

Now, by taking Eq. (2.86) into Eq. (2.84) and Eq. (2.85) we have that:

$$y_{\text{sonin-OML}}(R_p, \psi_p, \beta) = \frac{2}{\sqrt{\pi}} \sqrt{\frac{T_i}{T_e}\left(1 - \frac{e\varphi_p}{k_B T_i}\right)} \tag{2.87a}$$

$$x_{\text{sonin-OML}}(R_p, \psi_p, \beta) = \frac{2}{\sqrt{\pi}} \frac{r_p e^2 n_{e0}}{\varepsilon_0 k_B T_e} \sqrt{\frac{T_i}{T_e}\left(1 - \frac{e\varphi_p}{k_B T_i}\right)} \tag{2.87b}$$

and, if we consider the usual dimensionless parameters that we have been using across the document, we

have that:

$$y_{\text{sonin-OML}}(R_p, \psi_p, \beta) = \frac{2}{\sqrt{\pi}}\sqrt{\beta - \psi_p} \tag{2.88a}$$

$$x_{\text{sonin-OML}}(R_p, \psi_p, \beta) = \frac{2}{\sqrt{\pi}}R_p^2\sqrt{\beta - \psi_p} \tag{2.88b}$$

On the other hand, as the model has to be solved numerically, we do not have an analytical expression for the ion current collected by the probe in the ABR theory. Instead, we have to obtain the Sonin-plot from numerical $I - V$ characteristic curves, as the ones shown in Fig. 2.8b. So, it would be convenient to have expressions of the abscise and ordinate of the Sonin-plot in terms of the dimensionless current, I, that we defined in section 2.4.1 when solving the ABR equations. In Eq. (2.46), which gives us the definition of the dimensionless current I, we can solve for i to get:

$$i = I 2\pi k_B T_e \sqrt{\frac{2\varepsilon_0 n_{e0}}{m_i}} \tag{2.89}$$

And now, if we take Eq. (2.89) into Eq. (2.84) and Eq. (2.85), after some algebraic calculations, we have that:

$$y_{\text{sonin-ABR}}(R_p, \psi_p, \beta) = 2\sqrt{\pi}\frac{I}{R_p} \tag{2.90a}$$

$$x_{\text{sonin-ABR}}(R_p, \psi_p, \beta) = 2\sqrt{\pi} I R_p \tag{2.90b}$$

Finally, with the help of Eqs. (2.88) and Eqs. (2.90) we can obtain the Sonin-plot for both the OML and ABR models. But first, let us notice that, with the definitions given by Eq. (2.84) and Eq. (2.85), no matter which theory is considered, the explicit expression for the Sonin-plot is the same:

$$y_{\text{sonin}} = R_p^2 \, x_{\text{sonin}} \tag{2.91}$$

So, for a fixed R_p value, any the point of the Sonin-plot must fall in a straight line whose slope is given by the square of the dimensionless probe radius.

In Fig. 2.14 we can see the Sonin-plot corresponding to both the ABR and the OML theories. The thick solid line corresponds to the ABR model as given by Eqs. (2.90). Obviously, in the ABR model, the ratio of the ion to electron temperature is zero, $\beta = 0$. The biasing potential of the probe in this Sonin-plot is $\psi_p = -25$. As Eqs. (2.90) depend on the current collected by the probe, those values are found numerically by cross plotting of Fig. 2.8b for $\psi_p = -25$ and $R_p \in [0.5, 4]$. The thick dashed line corresponds to the OML model as given by Eqs. (2.88). In order to compare both models properly, the values of the parameters used to obtain the OML Sonin-plot have been also $\beta = 0$, $\psi_p = -25$ and $R_p \in [0.5, 4]$. Let us notice that the OML model is represented by a horizontal line in Fig. 2.14 because Eq. (2.88a) does not depend on R_p. Last but not least, the thin solid lines represent points in the Sonin-plot where the dimensionless probe radius is fixed, $R_p = 0.5$ and $R_p = 4$ respectively, as given by Eq. (2.91).

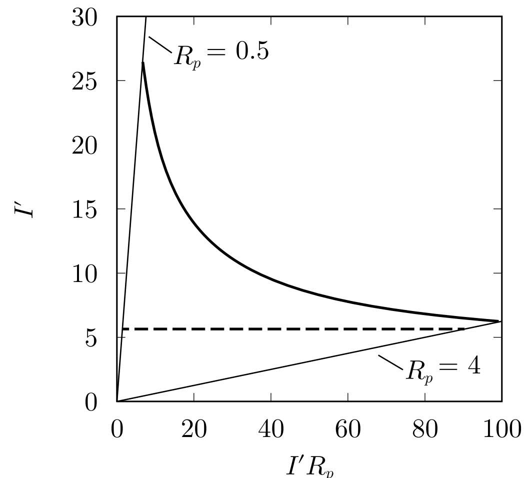


Figure 2.14: Sonin-plot for ABR (thick solid line) and OML (thick dashed line) for $\beta = 0$, $\psi_p = -25$ and $R_p \in [0.5, 4]$.

As can be seen in Fig. 2.14, the Sonin-plot of radial and orbital theories have very distinctive behaviours. In our case we have compared the ABR and OML models, but the results would be similar if we were to use any of the more elaborated models in each category. In Fig. 2.14, it can also be seen that, as the dimensionless probe radius is increased, the gap between both Sonin-plots is reduced. There are two reasons for this. On the one hand, the OML theory is only valid for the case $R_p \lesssim 1$, whereas for $R_p \gg 1$ the TSL is the correct approach. On the other hand, no matter what orbital model it is used, as R_p is increased it becomes more and more difficult

for an ion to orbit around the probe (see Fig. 2.4a), and the predictions of the orbital and radial models come closer to each other. Also, the larger the radius the more the probe behaves as planar instead of cylindrical. So both theories approach the planar limit as $R_p \to \infty$.

2.6. Conclusion

In this chapter we have reviewed the different models developed along the history that predict the ion current collected by a cylindrical probe, in the ion saturation zone. Two different approaches have been showcased: the orbital and the radial theories. Because of the profound differences between both cases, the predictions of the radial and orbital theories are very different, specially for relatively small probe radii, $R_p \lesssim 1$. We have stated that experimentalists have found results in agreement with either theories and even in between the predictions of both, the radial and orbital models. Moreover, it has been found a transition between radial and orbital behaviour for helium plasmas, which suggests that the radial and orbital behaviours should be limiting cases of a more general model that describes the movement of ions across the sheath. The Sonin-plot, which represents a powerful tool when it comes to discriminate between the radial and orbital behaviours, has also been explained.

With this chapter we finish the introductory part of the thesis. The following part is devoted to explain the details of the different simulations developed during our research. In particular, in the next chapter we are going to explain the fundamentals about particle-in-cell (PIC) simulations.

Part II

Particle Simulations

Chapter 3

Particle-In-Cell simulations & parallelisation techniques

3.1. Introduction

Because of the complex problems found when studying a plasma, computers have always played an important role in the development of plasma physics. To put it into perspective, even the simplest models developed in chapter 2, *e. g.* the ABR theory, require a numerical integration scheme in order to be solved. The only exception to that is the the Mott-Smith and Langmuir theory. This should come as no surprise, since the theory was developed in 1926, while the ENIAC, the first general purpose computer, was built in 1946.

When it comes to simulate a plasma with a computer, there are two main approaches: fluid description and kinetic description. Fluid description of a system relies on the use of macroscopic quantities, such as particle densities and flow velocities, instead of dealing with microscopic information, *e. g.* distribution function. Fluid simulations of plasmas are performed by solving magnetohydrodynamic (MHD) equations, which are composed by the different moments of the Boltzmann equation (see apendix A) and the field equations, *i. e.* Maxwell's equations. Actually, this is the approach used in chapter 2. Even thought the fluid approach offers a coarser description of the system, it was extensively used in the early ages of plasma physics, due to the fact that it is relatively inexpensive from the computational point of view. On the other hand, the kinetic approach relies on microscopic parameters such as distribution functions or even individual particle positions and velocities, reason why it offers a more detailed and complete description. Within the kinetic simulations there are two options, to solve the kinetic equations of plasmas (*i. e.* Boltzmann, Vlasov or Fokker-Planck equations along with field equations) or particle simulations. In particle simulations the motion of a collection of charged particles is computed while they interact with each other as well as with externally applied fields.

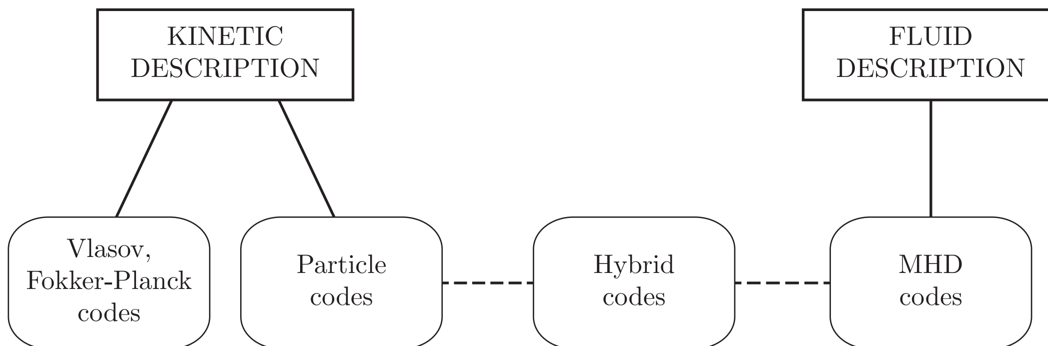


Figure 3.1: Classification of the different codes used to simulate plasmas.

In Fig. 3.1 we can see a simple classification scheme of the different options that can be used in order to simulate a plasma. MHD and Vlasov/Fokker-Planck codes consist basically in numerical algorithms

for solving differential equations, such as the RK4 algorithm that we have mentioned previously. On the other hand, particle codes are more elaborated pieces of software. Our research is based on the use of particle codes to simulate the contact of a plasma with a probe. In this chapter we are going to cover the theory behind those kind of simulations, as well as some of the details of the parallelisation technique that we have used to develop our codes.

3.2. Particle simulations and computer experiments

Let us start by defining what we meant by particle simulations. Computer simulations can be described in general as the numerical resolution of initial-values-boundary-values problems with the help of computers. **Particle simulation** is a generic term that describes a kind of simulation model where the description of the physical phenomena involves the use of interacting particles. That is, in computer simulations using particles, the system under study is represented by a certain ensemble of particles interacting according to certain rules. Here, the term "particle" must be understood in the most general way, however, in most applications the particles are identified with the physical objects that the system is composed of. Those physical objects can range from planets and starts, in astrophysical simulations, to ions and electrons, in plasma simulations, which is our case of study. Each particle has a set of attributes such as mass, charge, position, velocity, spin, etc. Many of those are defining magnitudes of the particles and, consequently, are constants along the simulation, while others are evolving quantities whose change is determined by the laws of interaction of the particles.

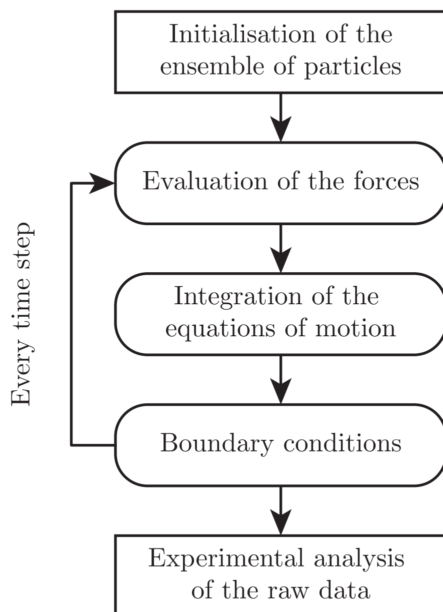


Figure 3.2: General scheme of any computer simulation using particles.

If we want to perform a particle simulation in a computer, the following elements have to be provided in the code of the program. First, some initial conditions for the ensemble of particles that conforms the system under study are needed. Then, the simulation needs rules to evaluate the dynamics of the particles present in the system, *i. e.* the forces acting on each particle. Also, appropriate equations of motion are needed for the simulation program in order to evaluate the motion of the particles. The existence of particles is usually limited to some finite region of the space, known as "computational box", on the limiting surface of which prescribed boundary conditions hold. So, proper boundary conditions need to be supplied as well.

Once that we have stablished the minimum information that is needed, in order to develop the code of a particle simulation, we can specify the very basic steps that are taken in every particle simulation from start to end. These steps can be seen summarised in the flow chart shown in Fig. 3.2. There we can see that, first, the ensemble of particles is initialised. After that, the program evaluates the force acting on every particle according to the rules prescribed in the code. Then the corresponding equations of motion are integrated for each particle, so, new values for the attributes associated with each particle are obtained, typically new positions and velocities. After that, it is posible that some of the particles have reached the boundaries of the computational box, so, this possibility has to be handled according to the boundary conditions implemented into the code. Next, the simulation goes back to the evaluation of the forces step and starts iterating over the code until the simulation finishes.

There is one last step that appears in the flow chart of Fig. 3.2 and that we have not already mentioned. This last step is the experimental analysis of the raw data produced by the simulation. Here is where the term, **computer experiment** arises. Let us notice that particle codes have more similarities, from a conceptual point of view, with an experiment than with other simulation approaches such as MHD, Vlasov or Fokker-Planck codes. In a particle simulation we have a system of particles

that behaves according to some mathematical model of the physical laws governing the system. Then, in order to obtain physically meaningful information, measurements have to be performed into the system. This task is done by analysing the raw data that the simulation manages, *i. e.* the value of all the attributes assigned to particles, for each particle in the system and for each time step in the simulation. Even the simplest particle simulation produces a huge amount of raw data, actually only positions and velocities by themselves constitute millions of values per time step. This data is then processed, averaged and analysed in order to obtain macroscopic and significant information, *e. g.* pressure, temperatures, electric currents, etc. This process is equivalent to the process of measuring any of those magnitudes in a physical experiment.

Computer experiments represent a powerful tool that complements both, theoretical and experimental research. Science has evolved by experiments that push the development of new theories in order to explain them, and by theories that predict situations that are then confirmed by experiments. Nowadays, computers have to be seen as a new "laboratory", where computer experiments are performed in order to bridge the gaps between both theory and experiments. This is achieved because a computer experiment is a completely controlled environment where, not even the process of measuring affects the system that is being measured, and the effect of fundamental laws of physics on complex systems can be studied. Some of the systems that take advantage of the use of computer experiments are shown in Table 3.1.

Particle simulations greatly depend on the system that is going to be simulated as well as the physical laws that it is governed by. However, they can be classified into three main categories, depending on how the evaluation of the forces suffered by every particle is carried out. Those categories are: particle-particle (PP) models, particle-mesh (PM) models and particle-particle–particle-mesh (PPPM or $\mathrm{P}^3\mathrm{M}$) models. Let us explain the difference between them.

Particle-Particle: From the conceptual point of view, PP models represent the simplest approach that can be followed to develop a particle simulation. PP models rely in the knowledge about the interaction between pairs of particles present in the system, *e. g.* the gravitational force between the different astral bodies in a planetary system. In this case, the way the force suffered by every particle in the system is evaluated, is rather simple. Let us call $\vec{f}_{i,j}(\vec{q_i}, \vec{q_j})$ to the force suffered by particle i due to particle j, which depends on the attributes of particle i, $\vec{q_i}$, as well as the attributes of particle j, $\vec{q_j}$. Then to obtain the net force suffered by particle i, we just have to evaluate the quantity:

$$\vec{F}_i(\vec{q_i}) = \sum_{\substack{j=1 \\ j\neq i}}^{N} \vec{f}_{i,j}(\vec{q_i}, \vec{q_j}); \quad \forall\ i \in [1, N] \tag{3.1}$$

N being the total number of particles present in the system. Let us notice that Eq. (3.1) has to be evaluated for each particle in the system.

The main advantages of these models are their simplicity and the fact that the force evaluation is carried out exactly, without any approximation or simplification. However, as counterpart, the computational resources needed for the evaluation of the forces, can easily overcome the performance of even the most up to date hardware. The number of operations needed to complete the force evaluation scales as $\mathcal{O}\left(N^2\right)$ in general, reason with the PP approach is only useful for finite systems which are relatively small, *e. g.* the solar system. Nevertheless, in case we are dealing with short-range interactions, *e. g.* the Lennard-Jones potential, where a cutoff distance can be defined for the interaction, every particle in the system would interact only with a certain number of close neighbours. In this case, the number of operations in the force evaluation no longer scales as $\mathcal{O}\left(N^2\right)$, and bigger systems or even infinite ones (by using cyclic contour conditions) can be simulated with a PP model.

Particle-Mesh: These models arise in order to overcome the limitations of the PP models, *i. e.* simulate systems where there are more than a few hundreds particles that interact through long-range forces. In order to avoid the costly evaluation of all the forces between pairs of particles, PM models rely on the evaluation of macroscopic fields instead of microscopic forces between particles. In these models, the computational box is meshed and, by means of the superposition principle, the macroscopic fields where the particles move are determined in the nodes of the mesh. Additionally, field equations could be required in PM models in order to evaluate de forces suffered by the particles, *e. g.* Maxwell's equations when electromagnetic interactions are present.

Example	Computer particles	Particle attributes	Physical			Computer model		
			N_p	L	T	N_p	L	T
1. *Correlated systems*								
Covalent liquids	Atoms or molecules	Strength constants related to quantum-mechanical dipole and quadrupole interactions, mass, force, position, velocity	10^5	10^{-8}	10^{-12}	$10^3 \sim 10^4$	$10^{-8} \sim 10^{-9}$	10^{-12}
Ionic liquids	Positive and negative ions	Charge, mass, force, position, velocity, radius	10^5	10^{-8}	10^{-12}	$10^3 \sim 10^4$	$10^{-8} \sim 10^{-9}$	10^{-12}
Stellar clusters	Stars	Mass, position, velocity, force, radius	$10^2 \sim 10^3$	10^{17}	10^{15}	$10^2 \sim 10^3$	10^{17}	10^{15}
Galaxy clusters	Galaxies	Mass, position, velocity, force, radius	$10^4 \sim 10^5$	10^{23}	10^{17}	$10^4 \sim 10^5$	10^{23}	10^{17}
2. *Collisionless systems*								
Collisionless plasma	"Superparticle" $\simeq 10^8$ electrons or ions	Charge, mass, position, velocity, radius	$10^9 \sim 10^{12}$	$10^{-5} \sim 10^{-2}$	$10^{-9} \sim 10^{-12}$	$< 10^5$	$10^{-5} \sim 10^{-2}$	$10^{-9} \sim 10^{-12}$
Galaxies – spiral structures	"Superparticle" $\simeq 10^6$ stars	Mass, position, velocity, radius	$10^{10} \sim 10^{11}$	10^{21}	10^{13}	$< 10^5$	10^{21}	10^{13}
3. *Collisional systems*								
Semiconductor devices (microscopic Monte-Carlo model)	"Superparticle" $\simeq 10^4$ electron wavepackets	Charge, mass, position, wavenumber, radius	10^8	10^{-7}	10^{-10}	$< 10^5$	10^{-7}	10^{-10}
4. *Collision-dominated systems*								
Semiconductor devices (diffusion model)	"Superparticle" $\simeq 10^4$ electrons or holes	Charge, position	10^9	10^{-6}	10^{-9}	$< 10^5$	10^{-6}	10^{-9}
Inviscid, incompressible fluids (vortex)	Vortex element	Vorticity, position	cont.	$10^{-3} \sim 10^6$	$10^{-3} \sim 10^5$	$< 10^5$	$10^{-3} \sim 10^6$	$10^{-3} \sim 10^5$

Table 3.1: Examples of physical systems where particle simulations and computer experiments make good sense. L is the characteristic length of the system, T its characteristic time and N_p is the number of particles in L^3. Credit: R. W. Hockney and J. W. Eastwood [50].

By evaluating macroscopic fields instead of microscopic forces between particles, these models achieve a much better performance from the computational point of view, and the number of operations needed for the evaluation of the forces of the particles scales as $\mathcal{O}(N\log(N))$ or even $\mathcal{O}(N)$. There are several algorithms to evaluate the macroscopic fields, and which one is used depends mainly on the kind of interaction that is going to be simulated and the ratio of exactitude to performance that it is going to be assumed. Among these we can find the Particle-In-Cell algorithm, which is the one that we are going to use in our simulations.

Particle-Particle–Particle-Mesh: $\mathrm{P^3M}$ models kind of have the best of both, PP and PM models. The main problem of PP models is that the number of operations needed to evaluate de forces can easily become cumbersome, while the problem of PM models is that they tend to neglect the effect of very short range interactions between particles, *i. e.* collisions. $\mathrm{P^3M}$ models are based on a PM model, but the interaction of each particle with their closest neighbours within a cutoff radius are also taken into account. In this way, the force suffered by each particle has two contributions, one due to the macroscopic field, evaluated through the mesh, and the other due to the microscopic interaction with other particles within a certain distance.

$$\vec{F}_i(\vec{q}_i) = \vec{f}_{i\text{-mesh}} + \sum_{\substack{j=1 \\ j\neq i \\ |\vec{r}_{ij}|\leq r_{\text{cutoff}}}}^{N} \vec{f}_{i,j}(\vec{q}_i, \vec{q}_j); \quad \forall\, i \in [1, N] \tag{3.2}$$

where $\vec{f}_{i\text{-mesh}}$ is the force suffered by the i-th particle due to the macroscopic fields, $|\vec{r}_{ij}|$ is the distance between particle i and particle j and r_{cutoff} is the distance at which short-range interactions start to be neglected.

Even though in Eq. (3.2) we have expressed the short range interactions as a sum of forces between pairs of particles, there are several algorithms to evaluate them that tackle this problem by different approaches, *e. g.* Monte Carlo (MC) methods.

The force evaluation probably represents the trickiest part of any particle simulation, while the integration of the equations of motion is usually a rather simple task. Depending on the velocities of the particles present in the system, particle simulations can be: relativistic and non-relativistic. Let us describe both possibilities a little further.

Non-relativistic: When considering the case of non-relativistic systems, *i. e.* systems where particles move with small velocities when compared to c, once the force suffered by every particle is known, all that has to be done is to integrate Newton's second law for each particle:

$$\vec{F}_l = \frac{d\vec{p}_l}{dt} = \frac{dm_l\vec{v}_l}{dt} = m_l\frac{d^2\vec{r}_l}{dt^2}; \quad \forall\, l \in [1, N] \tag{3.3}$$

where $\vec{p}_l$ is the momentum of the l-th particle, m_l its mass, $\vec{v}_l$ its velocity and $\vec{r}_l$ its position. Eq. (3.3) is easily integrated numerically with a finite difference approximation. Depending on the precision required, different algorithms and schemes are available.

Relativistic: Contrary to the non-relativistic case, when the velocity of particles present in the system is on the same order of magnitude as c, relativistic effects must be taken into account during their motion. For this reason, the relativistic expression of the momentum of the particles, $\vec{p}_l = \gamma(v_l)m_l\vec{v}_l$, has to be used in Eq. (3.3), where:

$$\gamma(v_l) = \frac{1}{\sqrt{1 - \frac{v_l^2}{c^2}}} \tag{3.4}$$

Once we introduce Eq. (3.4) into Eq. (3.3), all that remains is to integrate numerically the equation of motion with a finite difference algorithm as in the non-relativistic case.

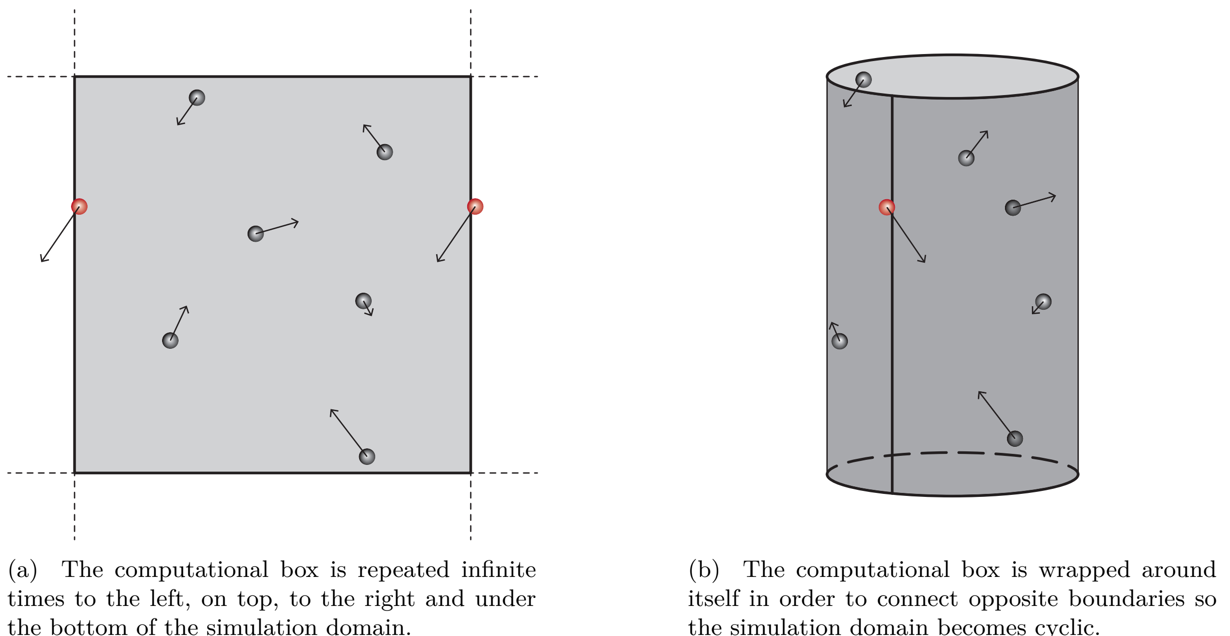
Finally, we can also establish a classification of the different particle simulations depending on the boundary conditions stablished. In this sense, the walls limiting the computational box of a particle simulation can be: absorbing, reflecting, emitting or cyclic boundaries. Obviously, in a particle simulation we can have any mixture of them. Let us characterise a little bit the aforementioned boundaries.

Absorbing boundaries: Particles that cross an absorbing boundary are withdrawn from the computational box and thus from the simulation itself. Along with other mechanisms, such as particle annihilation, absorbing boundaries represent one of the means that allow a simulation to decrease the number or particles within it.

Emitting boundaries: Contrary to the previous one, an emitting boundary emits particles that are then injected into the simulation. When absorbing boundaries are present in the computational box, emitting ones (or any other source of particles) needs to be implemented into the simulation. Otherwise the simulation would end up empty.

Reflecting boundaries: When a closed system is going to be simulated, it usually needs some degree of reflectiveness in their walls. The law that govern the behaviour of a reflective boundary does not need to be specular reflection. Instead, different reflection rules can be implemented, in order to take into account the physical properties of the simulated wall.

Cyclic boundaries: There are situations when it results of great interest to simulate an infinite systems. Obviously, the problem being that, in order to do that, we would need infinite computational resources. One of the simplest ways to deal with this problem is to simulate a finite system with cyclic boundaries. Cyclic boundaries are defined by pairs of walls that are kind of "connected". A graphic representation of a cyclic boundary can be seen in Fig. 3.3. By using this boundaries the particles, virtually, never find a wall in their movement. Under this conditions, the simulation behaves as if it were infinite, as long as the computational box is large enough to avoid correlations between particles in the extremes of it.



(a) The computational box is repeated infinite times to the left, on top, to the right and under the bottom of the simulation domain.

(b) The computational box is wrapped around itself in order to connect opposite boundaries so the simulation domain becomes cyclic.

Figure 3.3: Two different interpretations of a cyclic boundary in a two dimensional domain.

We have to notice that, simulations do not have to fit exactly into one of the previous classifications in terms of: the force evaluation, the equations of motion and the boundary conditions. Some simulations consider relativistic equations of motion for one kind of particle and non-relativistic for others, or have different boundary conditions on different walls of the computational box, or even have different force evaluation methods for different kind of particles. In the following section we will discuss more in depth the exact simulations that we are going to use in our research.

3.3. The Particle-In-Cell method

As any other particle code, PIC simulations follow the scheme described in Fig. 3.2. In this section we are going to explain, in detail, the three parts that represent the core of any PIC simulation, that

is: the force evaluation, the integration of the equations of motion and the boundary conditions. Let us start by remembering that we are interested in simulating the contact of a collisionless plasma with a perfectly absorbing Langmuir probe that is negatively biased with respect to the plasma. So, in order to simulate this system, the following considerations must be taken into account:

- Due to the huge number of particles interacting with electrostatic potentials (long range), we are going to use a PM model. As collisions are not going to be taken into account, we do not need to develop a P^3M model. Also, we have to take into account Poisson's equation to obtain the electric potential distribution in the mesh.
- As the velocities involved when the ions and electrons approach the surface of the probe are far from being relativistic, we are going to use non-relativistic equations of motion, *i. e.* Eq. (3.3).
- We are going to simulate the space between the surface of the probe and the plasma. The boundary corresponding to the surface of the probe is going to be, obviously, perfectly absorbing, in order to represent the perfectly absorbing probe. Although, the boundary corresponding to the plasma is going to be both, absorbing and also emitting, as it has to represent an imaginary surface, far away from the probe, so that particles cross from the probe towards the plasma (absorbing) and from the plasma towards the probe (emiting).

What we have just described is a collisionless, non-relativistic, electrostatic Particle-In-Cell (PIC) simulation of the contact of a plasma with a Langmuir probe. Even though the name **Particle-In-Cell** corresponds to one of the algorithms used during the force evaluation, as we will see, "PIC simulation" has become a usual term to describe almost any PM model that considers electromagnetic interactions between particles. With such a large scope, it is not our intention to cover here a complete description of the theory behind PIC simulations, but to give a general description of the main algorithms particularised for the simulations that we have developed. For a more complete and detailed description of PIC techniques the reader is referred to the classical monographs written by R. W. Hockney and J. W. Eastwood [50], and C. K. Birdsall and A. B. Langdon [51].

3.3.1. Force evaluation

The force evaluation is the part of the code that set apart PIC simulations from any other simulation using particles. As we have previously stablished, PIC simulations are based in PM models where particles interact with each other through electromagnetic fields.

We have to notice that, in order to properly simulate a plasma, the electromagnetic fields must be self-consistently evaluated. That is, the electric and magnetic fields must be obtained from quantities that depend on the particles configuration at the moment of solving them. As it is known, the sources of electromagnetic fields are charge and currents densities. So, before solving Maxwell's equations, charge and currents densities must be obtained at the nodes of the mesh from the particle attributes.

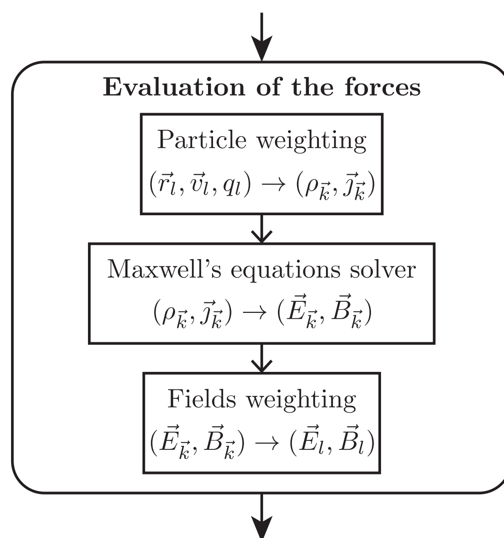


Figure 3.4: Force evaluation in PIC codes. l index refers to particles, while $\vec{k}$ index refers to nodes.

In Fig. 3.4 the general scheme of the force evaluation in PIC codes is shown. There, we can see that there are three main parts involved in the evaluation of the forces: the particle weighting, the solution of Maxwell's equation and the fields weighting. However, as our study is centred in the case of unmagnetised plasmas, magnetic interactions are considered to be negligible, and from now on we are going to consider only electrostatic interactions. We made the same assumption in the previous chapters when reviewing the fluid models. The difference with the

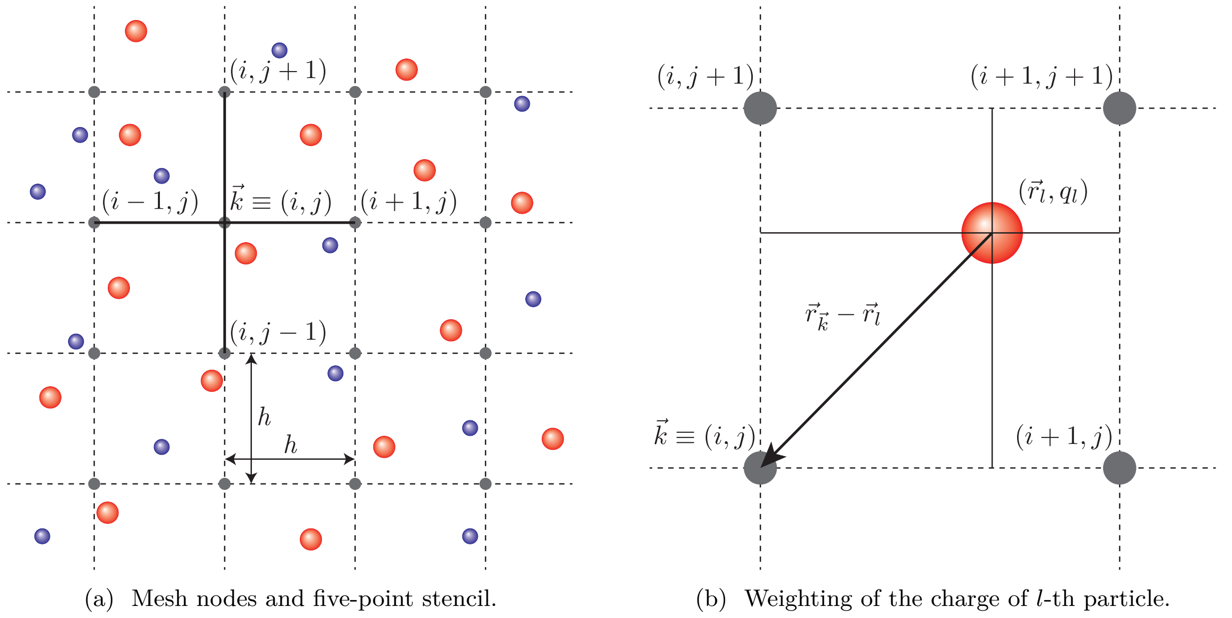
scheme shown in Fig. 3.4 is that instead of solving the full set of Maxwell's equations, we only have to solve Poisson's equation:

$$\nabla^2 \varphi(\vec{r}) = -\frac{\rho(\vec{r})}{\varepsilon_0} \tag{3.5}$$

But, prior to the numerical resolution of Eq. (3.5), it is necessary to grid the space, *i. e.* $\vec{r} \to \vec{r}_{\vec{k}}$. Where $\vec{k}$ represents the cartesian coordinates of the nodes of the mesh, *e. g.* in 2D we have that $\vec{k} \equiv (i,j)$ and $\vec{r}_{\vec{k}} \equiv \vec{r}_{i,j}$, as seen in Fig. 3.5. By following with the two dimensional case for the sake of clarity, once we have defined a mesh in the computational box, Poisson's equation can be discretized by using a finite difference scheme (five-point stencil) as follows:

$$\frac{\varphi(\vec{r}_{i-1,j}) + \varphi(\vec{r}_{i+1,j}) + \varphi(\vec{r}_{i,j-1}) + \varphi(\vec{r}_{i,j+1}) - 4\varphi(\vec{r}_{i,j})}{h^2} \simeq \frac{\rho(\vec{r}_{i,j})}{\varepsilon_0} \tag{3.6}$$

h being the spacing of the mesh, which is supposed to be the same in any direction.



(a) Mesh nodes and five-point stencil.

(b) Weighting of the charge of l-th particle.

Figure 3.5: Particle weighting in a 2D PIC simulation.

In Eq. (3.6) we can see that, in order to solve Poisson's equation, the charge density at the position of the nodes, $\rho(\vec{r}_{\vec{k}}) \equiv \rho(\vec{r}_{i,j})$, is needed. Here is where the need of the particle weighting step arises. The particle weighting part of a PIC simulation, handles the obtention of an approximate value of the charge density at node positions, that is $\rho_{\vec{k}} \simeq \rho(\vec{r}_{\vec{k}})$, or in the two dimensional case $\rho_{i,j} \simeq \rho(\vec{r}_{i,j})$. In order to obtain this approximation, particles are assumed to have a certain "shape" or "size", depending of which the charge of each particle is "shared" across one or more of its surrounding nodes. In this way, the contribution of the l-th particle to the charge density of the $\vec{k}$-th node can be expressed as:

$$\rho^l_{\vec{k}} = \frac{q_l}{V_c} S\left(\vec{r}_{\vec{k}} - \vec{r}_l\right) \tag{3.7}$$

where q_l is the charge and $S\left(\vec{r}_{\vec{k}} - \vec{r}_l\right)$ the shape function associated with the l-th particle. Also, V_c is the volume occupied by an individual "cell"[1] of the mesh. So the complete charge density associated with the $\vec{k}$-th node is:

$$\rho_{\vec{k}} = \sum_{l=0}^{N} \frac{q_l}{V_c} S\left(\vec{r}_{\vec{k}} - \vec{r}_l\right) \tag{3.8}$$

Let us notice that, the stability and the speed of a PIC simulation greatly depends on the exact shape function considered. For instance, let us consider two limiting cases:

[1] "Cell" is the name that it is usually given to the subdomain of a PIC simulation defined by adjacent nodes.

- On the one hand, if the shape function is wide enough so that every particle contributes to the charge density of every node in the mesh, the number of operations needed for the evaluation of the forces does not differ so much from the PP models. For this reason, shape functions for which each particle is only weighted to a certain small number of close nodes are preferred.
- On the other hand, if we consider the limiting case of assigning the whole charge of a particle to its nearest grid point, the charge density associated to the nodes would change drastically as particles move. This would lead to a noisy charge density and, ultimately, to noisy potential and field distributions, affecting the stability of the simulation.

So, when looking for a shape function, a compromise must be reached between speed of the simulation and smoothness/stability of the electric potential and field. As we are going to see, this compromise is found in terms of the maximum number of close nodes to which each particle is weighted. We must notice that, the functional dependence of the shape functions, $S(\vec{r}_{\vec{k}}-\vec{r}_l)$, can not be whatever we want, instead, it has to satisfy certain conditions.

The first two conditions are rather simple. First, due to the space isotropy, we have that shape functions must satisfy:

$$S\left(\vec{r}_{\vec{k}}-\vec{r}\right)=S\left(\vec{r}-\vec{r}_{\vec{k}}\right) \tag{3.9}$$

And due to the charge conservation constraints, it also has to satisfy:

$$\sum_{\vec{k}} S\left(\vec{r}_{\vec{k}}-\vec{r}\right)=1 \tag{3.10}$$

The rest of the conditions that shape functions must satisfy, can be obtained by an increasing accuracy of the weighting scheme. For the sake of simplicity, let us consider the monodimensional case in order to obtain these conditions. We start by considering the potential at position x due to a unit charge located at the point X. Such a potential is given by the Green's function, $G(x-X)$. Now, if we introduce the weighting scheme, the potential at point x, created by a particle of charge q located at X, can be written as:

$$\varphi(x)=q\sum_{k=1}^{m} S(x_k-X)G(x-x_k) \tag{3.11}$$

m being the number of nearest nodes to which the particle is weighted. If we Taylor expand $G(x-x_k)$ around $(x-X)$ we have that:

$$\varphi(x)=q\sum_{k=1}^{m} S(x_k-X)G(x-X)+q\sum_{k=1}^{m} S(x_k-X)\sum_{n=1}^{\infty}\frac{(X-x_k)^n}{n!}\frac{d^nG(x-X)}{dx^n} \tag{3.12}$$

and by taking Eq. (3.10) into the first term on the right hand side of Eq. (3.12)

$$\varphi(x)=qG(x-X)+q\sum_{n=1}^{\infty}\frac{1}{n!}\frac{d^nG(x-X)}{dx^n}\sum_{k=1}^{m} S(x_k-X)(X-x_k)^n=qG(x-X)+\delta\varphi(x) \tag{3.13}$$

We can see that, the first term in the right hand side of Eq. (3.13) represents, precisely, the potential at x due to a particle of charge q located at X. So the term $\delta\varphi(x)$ is an unphysical potential introduced by the weighting scheme. Obviously, it is desirable this term to be as small as possible, which can be done by requiring:

$$\sum_{k=1}^{m} S(x_k-X)(X-x_k)^n=0 \tag{3.14}$$

for $n=1,\ldots,n_{\max}-1$, $n_{\max}-1$ being the last value of n for which Eq. (3.14) is verified. And by taking Eq. (3.14) into Eq. (3.13), we have that:

$$\begin{aligned}\delta\varphi(x)&=q\sum_{n=n_{\max}}^{\infty}\frac{1}{n!}\frac{d^nG(x-X)}{dx^n}\sum_{k=1}^{m} S(x_k-X)(X-x_k)^n\\&\sim G(x-X)q\sum_{k=1}^{m} S(x_k-X)\sum_{n=n_{\max}}^{\infty}\frac{(X-x_k)^n}{n!(x-X)^n}\end{aligned} \tag{3.15}$$

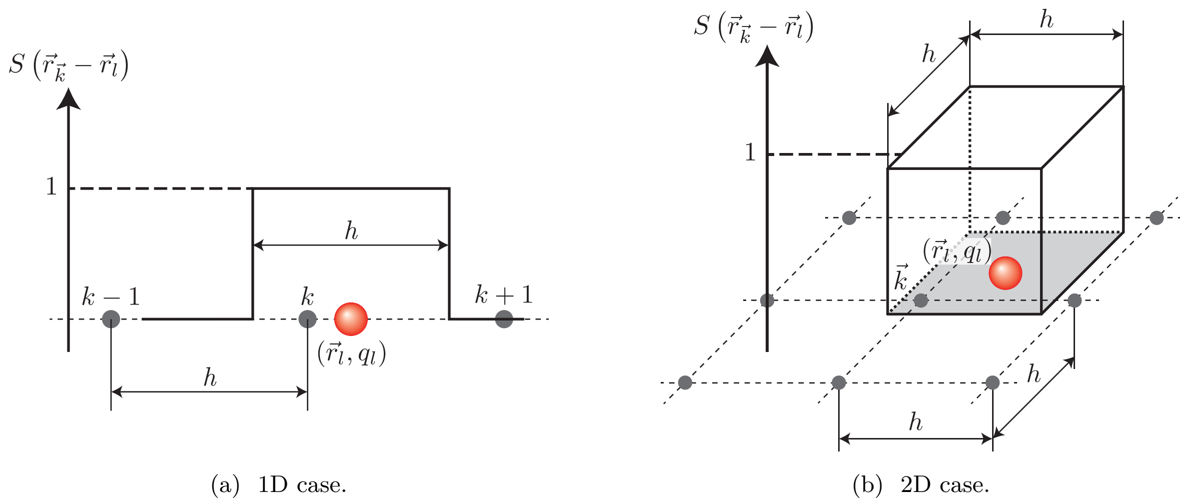
Thus, at large distances, $|X - x_k| < |x - X|$, we have that $\delta\varphi(x)$ decreases as $n_{\max}$ increases.

Finally, the shape functions can be obtained by taking into account the conditions given by Eq. (3.9), Eq. (3.10) and Eq. (3.14). It has to be noticed that, the value of $n_{\max}$ depends on the maximum number of closest nodes, m, to which the particle is weighted. Now, we are going to see the two simplest weighting schemes, which are also the two most used along the history of PIC simulations.

NGP: This is the simplest scheme to obtain the charge density at node positions from the particles configuration. We even have already briefly mentioned this scheme, and it consist of assigning the whole charge of the particle to its nearest node. For this reason, it is usually called Nearest-Grid-Point (NGP). The shape function corresponding to this scheme can be defined as:

$$S^0(\vec{r} - \vec{r}_l) = \begin{cases} 1 \text{ , if } |\vec{r}_k - \vec{r}_l| < \frac{h}{2} \\ 0 \text{ , otherwise} \end{cases} \tag{3.16}$$

As we mentioned before, this scheme generates very noisy simulations, reason why its use is not recommended. However, due to the very low computational resources that it needs, it was extensively used in the early ages of PIC simulations. In Fig. 3.6 it can be seen the representation of the shape function of the NGP scheme for the 1D and 2D cases.



(a) 1D case.

(b) 2D case.

Figure 3.6: Nearest-grid-point (NGP) weighting scheme.

CIC / PIC: This first order weighting scheme, assigns charge density, not only to the closest node of a particle, but to all the nodes that delimit the cell where the particle is. So, the number of nodes to which every particle is weighted is: 2 in the 1D case, 4 in the 2D case and 8 in the 3D case. The shape function corresponding to this scheme in the 1D case is:

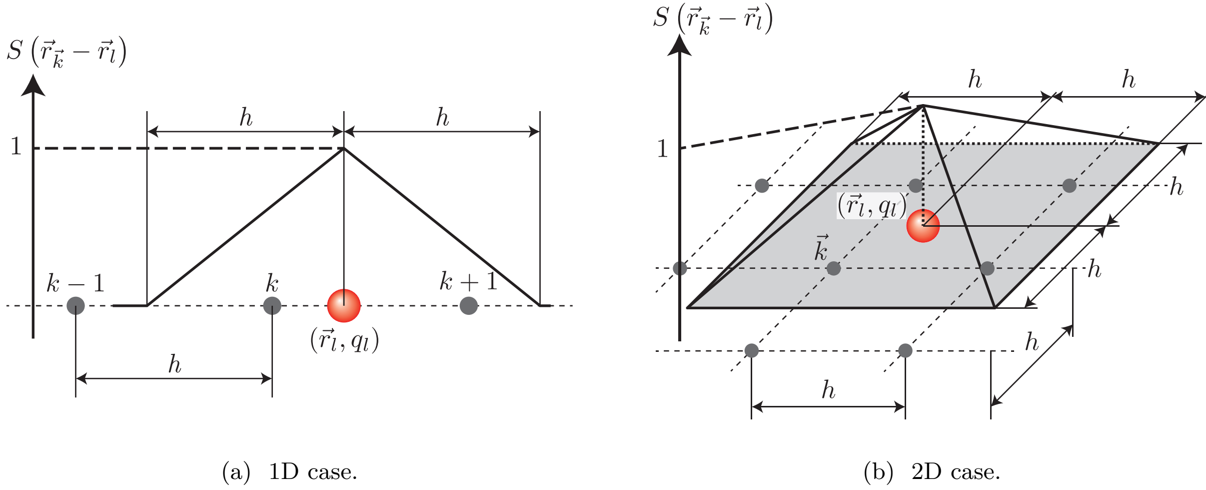
$$S^1(x - x_l) = \begin{cases} 1 - \dfrac{|x - x_l|}{h} \text{ , if } |x - x_l| < h \\ 0 \text{ , otherwise} \end{cases} \tag{3.17}$$

However, Eq. (3.17) can be generalised easily for the case of multidimensional cartesian coordinates as:

$$S^1(\vec{r} - \vec{r}_l) = S^1(x - x_l)\, S^1(y - y_l)\, S^1(z - z_l) \tag{3.18}$$

This scheme is usually referred to as Cloud-In-Cell or CIC. Because of the form of the shape function, the particle can be seen as a uniformly charged cloud of width h, whose differential charges contribute to their nearest grid point. An alternative view is that, every particle resides in a cell and, consequently, contributes proportionally to the nodes delimiting that cell. From this interpretation the term Particle-In-Cell or PIC arises. It should come at no surprise that, the kind of simulations we are dealing with are referred to as PIC simulations, since this is the most widely used weighting scheme.

In Fig. 3.7 it can be seen the representation of the shape function of the CIC/PIC scheme for the 1D and 2D cases. There we can see that this scheme is based on the use of "triangular" function shapes.



(a) 1D case.

(b) 2D case.

Figure 3.7: Cloud-in-cell (CIC) / Particle-in-cell (PIC) weighting scheme.

Higher order weighting by using quadratic and cubic splines rounds off the roughness in the particle shape function and, as a result, reduce even further the noise in all the macro-quantities. Also, it can be probed that, by using higher order schemes, the appearance of nonphysical effects in the simulations is diminished. However, this is achieved at the expense of a computationally much heavier simulation. The first order weighting scheme, kind of resides in the sweet spot. Even though CIC/PIC schemes consume more computer resources than NGP, for a given noise level, they allow both a coarser grid and fewer particles than NGP, and thus regains some of the additional computer time required per particle.

Once the approximation to the charge density at node positions, $\rho_{\vec{k}}$, is obtained, the discretized Poisson's equation can be numerically solved. In this way, an approximation of the electric potential at node positions would be obtained, *i. e.* $\varphi_{\vec{k}} \simeq \varphi(\vec{r}_{\vec{k}})$. As we already introduced, by using a finite difference approximation, we can transform the resolution of Poisson's equation in the simulation domain (ODE) into the resolution of the following system of linear equations:

$$\sum_{\vec{n}} c_{\vec{n}} \varphi_{\vec{n}} = -\frac{h^2 \rho_{\vec{k}}}{\varepsilon_0} \tag{3.19}$$

where $\varphi_{\vec{n}}$ are the unknowns. Eq. (3.19) has to be particularized depending on the dimensionality considered and the order of the finite difference approximation of the Laplace operator in Eq. (3.5). For second order approximations in 1D and 2D the resulting systems are:

$$\varphi_{i-1} - 2\varphi_i + \varphi_{i+1} = -\frac{h^2 \rho_i}{\varepsilon_0}; \quad \forall i = 1, \ldots, N_x \ ; \text{(1D)} \tag{3.20a}$$

$$\varphi_{i-1,j} + \varphi_{i,j-1} - 4\varphi_{i,j} + \varphi_{i+1,j} + \varphi_{i,j+1} = -\frac{h^2 \rho_{i,j}}{\varepsilon_0}; \quad \forall i = 1, \ldots, N_x \ ; \ \forall j = 1, \ldots, N_y \ ; \text{(2D)} \tag{3.20b}$$

N_x and N_y being the number of nodes that the simulation has in either direction.

The systems given by Eqs. (3.20) can be easily solved with different numerical algorithms, in order to obtain the electric potential at any node of the computational box. At the end, the specific numerical algorithm used depends on a variety of factors. In our case, as we are interested in the parallel execution of our codes, the resulting system of linear equations is solved by using a Jacobi method. Jacobi method is simple and readily parallelizable, with a large fraction of its code being 100% parallel, which makes it perfect for its execution in Graphics Processing Units.

Once the system given by Eq. (3.19) is solved, no matter the algorithm used, it is straightforward to obtain the electric field a the node positions. Knowing that:

$$\vec{E}(\vec{r}) = -\vec{\nabla}\varphi(\vec{r}) \tag{3.21}$$

we only need to perform numerically the derivative of the electric potential. If we use second order approximations in 1D and 2D, the electric fields would be obtained as:

$$E_{x\,i} = \frac{\varphi_{i-1} - \varphi_{i+1}}{2h} \ ; \text{(1D)} \tag{3.22a}$$

$$E_{x\,i,j} = \frac{\varphi_{i-1,j} - \varphi_{i+1,j}}{2h} \ ; \quad E_{y\,i,j} = \frac{\varphi_{i,j-1} - \varphi_{i,j+1}}{2h} \ ; \text{(2D)} \tag{3.22b}$$

Once the fields are determined at node positions, the last step that has to be taken in order to complete the force evaluation in PIC codes is the field weighting. To a certain extent, this last step is exactly the opposite to the first one, the particle weighting. During the particle weighting, the value of microscopic quantities associated with particles, q_l, are interpolated in order to obtain macroscopic quantities associated with the nodes of the mesh, $\rho_{\vec{k}}$. While during the field weighting, the value of macroscopic quantities associated with the nodes of the mesh, $\vec{E}_{\vec{k}}$, are interpolated in order to obtain microscopic quantities associated with the particles, $\vec{E}_l$.

When it comes to choose a field weighting scheme, there are two main options: momentum conserving-schemes and energy-conserving schemes. In our case, we decided to use a momentum-conserving scheme. That is, a scheme where, in the absence of roundoff errors, the total momentum is identically conserved. It can be probed [52] that, the total momentum is conserved by any particle mesh calculation which fulfil the following conditions:

1. Identical particle weighting and field weighting schemes are used.
2. Correctly space-centred difference approximations to derivatives are used.

On the one hand, let us notice that, we have already fulfilled the second condition. All the derivatives approximations that we have used at this point, *e. g.* Eq. (3.6), Eqs. (3.20) and Eqs. (3.22), are properly space-centred and second order accurate with respect to the spatial step, *i. e.* errors scale as $\mathcal{O}\left(h^2\right)$. The reason behind the use of space-centred approximations is not only that we are interested in the use of a momentum-conserving scheme, since the use of properly space-centred approximations is preferred whenever possible over forward or backward approximations.

On the other hand, in order to obtain a momentum-conserving scheme, we only have to use the same shape functions that were used during the particle weighting in the field weighting. So, similarly to Eq. (3.8), the field at particle positions is found by using the following expression:

$$\vec{E}_l = \sum_{\vec{k}} \vec{E}_{\vec{k}} S\left(\vec{r}_l - \vec{r}_{\vec{k}}\right) \tag{3.23}$$

where the exact same shape function than in Eq. (3.8) should be used. In our case, this will be the provided by the CIC/PIC scheme, given in Eq. (3.17) and Eq. (3.18).

By using a momentum-conserving scheme, and thus fulfilling the previously mentioned conditions, we are ensuring that in our PIC simulations, self-forces are null and inter-particle forces are equal and opposite. That is, that particles do not apply any force to themselves and that the third Newton's law is fulfilled.

Once the field weighting is finished, the force evaluation step of the PIC simulation is completed, since the force suffered by the l-th particle is simply evaluated as:

$$\vec{F}_l = q_l \vec{E}_l \tag{3.24}$$

In the next section we are going to explain the details involved in the integration of the equations of motion.

3.3.2. Integration of the equations of motion

The integration of the equations of motion is a fundamental part, not only in PIC codes, but in any particle simulation. The part of the code that handles this task is usually called "particle mover". We

have to notice that, during a PIC simulation, the trajectory of each individual particle in the system is followed. For this reason, and knowing the large number of particles ($\gtrsim 10^5$) present in a PIC simulation, the two main traits that are appreciated in a particle mover are: high accuracy and speed. Just as in the force evaluation method, it is not possible to increase the accuracy without reducing the speed of the simulation, so the sweet spot between these two constraints has to be found.

As we stated at the beginning of Section 3.3, our simulations are going to fall in the non-relativistic velocity regime. So, the equation of motion that we have to solve is the one shown in Eq. (3.3). Let us notice that, by considering Eq. (3.24), Eq. (3.3) can be rewritten in the following form:

$$\frac{d\vec{r}_l}{dt} = \vec{v}_l \tag{3.25a}$$

$$\frac{d\vec{v}_l}{dt} = \frac{q_l}{m_l}\vec{E}_l \tag{3.25b}$$

Exactly as we did with the field equations in the previous section, in order to numerically solve the time derivatives that appear in Eqs. (3.25), a finite difference scheme is used. In order to proceed this way, first, the time in the PIC simulation should be divided into discrete steps, *i. e.* the time should be discretized just as the space. Let us remember that, PIC codes are iterative particle simulations (see Fig. 3.2), where different physical magnitudes are evaluated at discrete instants corresponding to the different iterations of the code. Usually, the time step, Δt, is constant along the simulation, so that the simulated time at the p-th iteration is given as: $t^{\,p} = p * \Delta t$. Accordingly, the value of any magnitude which depends on the time, $A(t)$ is mapped into discrete values at the corresponding times, $A(t) \to A^{\,p} = A(t^{\,p})$. For instance, the electric field acting on the l-th particle appearing in Eq. (3.24) and Eq. (3.25b), actually represents the electric field evaluated at the p-th iteration of the PIC simulation, so:

$$\vec{E}_l \equiv \vec{E}_l(t^{\,p}) = \vec{E}_l^{\,p} \tag{3.26}$$

For the discretization of Eqs. (3.25), we have used the **leap-frog** scheme, which is probably the most widely used scheme for the integration of the equations of motion. The leap-frog method relies on the use of time centred approximations, of the derivatives in Eqs. (3.25), for achieving good performance with a small computational footprint. In order to have centred approximations for both, the position, Eq. (3.25a), and the velocity, Eq. (3.25b), the method does not calculate particle velocities at usual instants, *i. e.* $t^{\,p}$. Instead, velocities are evaluated at half time steps, *i. e.* $t^{\,p+1/2} = (p + 1/2)\Delta t$. So, once discretized, the equations of motion would be:

$$\frac{\vec{r}_l^{\,p+1} - \vec{r}_l^{\,p}}{\Delta t} = \vec{v}_l^{\,p+1/2} \Rightarrow \vec{r}_l^{\,p+1} = \vec{r}_l^{\,p} + \Delta t\ \vec{v}_l^{\,p+1/2} \tag{3.27a}$$

$$\frac{\vec{v}_l^{\,p+1/2} - \vec{v}_l^{\,p-1/2}}{\Delta t} = \frac{q_l}{m_l}\vec{E}_l^{\,p} \Rightarrow \vec{v}_l^{\,p+1/2} = \vec{v}_l^{\,p-1/2} + \Delta t\ \frac{q_l}{m_l}\vec{E}_l^{\,p} \tag{3.27b}$$

As it can be seen in Eqs. (3.27), the leap-frog method is an explicit solver. This means that the values of the quantities that are being actualized depend only on values at older time steps, which are already known. In Fig. 3.8 a graphical representation of the method can be seen.

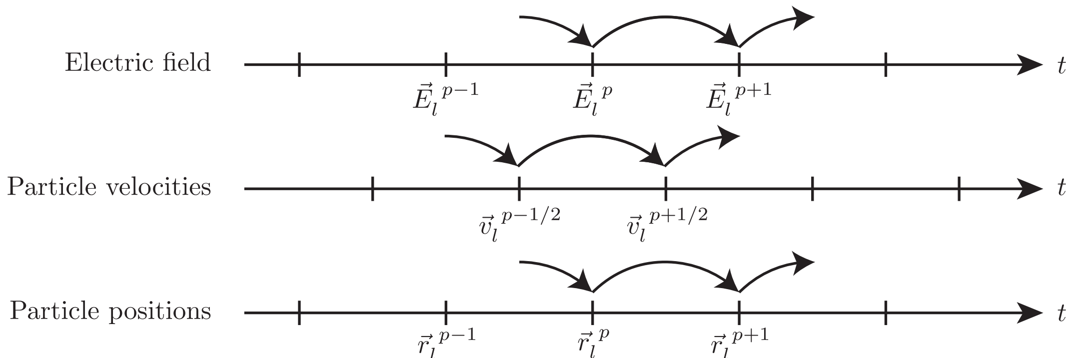


Figure 3.8: Graphical visualisation of the leap-frog integration scheme.

We have to notice that, due to the leap-frog scheme shown in Fig. 3.8, the particle velocities have to be shifted half time step when initiating the simulation. So, this has to be considered when developing the initialization step shown in Fig. 3.2.

3.3.3. Boundary conditions

The last part of PIC simulations that has to be discussed is the boundary conditions module. Here, we are going to consider only the cases of: absorption and injection of particles by the boundaries, since those are the boundary conditions that we are going to implement in our simulations. From the physical point of view, the aforementioned boundary conditions are rather simple: particles can be absorbed at any boundary or injected from them with any distribution function.

Absorption of particles by a boundary is trivial to implement into the simulations. Particles are withdrawn from the simulation as soon as they cross the corresponding wall. Nevertheless, from the computational point of view, injection of particles can be a little bit tricky. The two main problems are that:

1. Positions and velocities of particles are shifted in time $\Delta t/2$, as we have just said when discussing the particle mover.
2. Positions and velocities of particles are known at discrete time steps, while particle can cross the boundaries of the simulation at any moment.

The way this problems are handled is by introducing an extra particle mover step with an adjustable time step. By doing so, we can adjust the properties of the particles from the instant when they enter into the computational box to the proper time steps of the simulation. So, all that remains is to evaluate the flux of particles that the simulation is going to be fed with.

Let us start by taking into account that the boundary that our simulation is going to be fed through, should represent the sheath edge. So, the particles entering the simulation should have the properties of the quasineutral zone that we talked about in Section 1.4.1. In that zone, particles are described by the following distribution function:

$$f(\vec{v}) = n\left(\sqrt{\frac{m}{2\pi k_B T}}\right)^3 \exp\left(-\frac{m}{2k_B T}(v_x^2 + v_y^2 + (v_z - v_d)^2)\right) \tag{3.28}$$

n being the density of particles, m the particle mass, T the particles temperature and v_d a drift velocity.

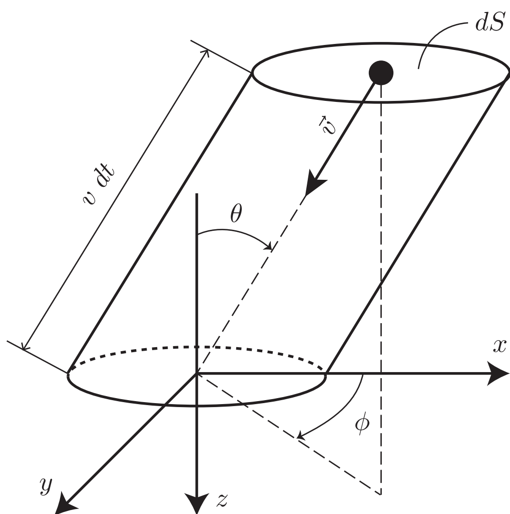


Figure 3.9: Diagram to evaluate the flux of particles through a wall.

We have to notice that, the previous expression, has not been particularized for any kind of particle, as any of them can be described by it. The drift velocity included in the distribution, takes account of the potential drop that takes place from the plasma to the presheath. This potential drop can be seen for the case of a planar probe in the low ionization limit in Fig. 1.7. We have to say that, when describing electrons, the drift velocity must be set to zero. As we are considering the case o negatively biased probes, electrons are repelled by it and, thus, they are not accelerated but retarded through the presheath.

In order to evaluate the flux of particles through a wall, let us start by thinking in the number of particles with velocities $\vec{v} \in [\vec{v}, \vec{v} + \vec{dv}]$. These particles would have suffered a displacement $\vec{v}\ dt$, a time dt later. Now, if we consider a differential surface area, dS, in the xy plane, every particle with velocities $\vec{v} \in [\vec{v}, \vec{v} + \vec{dv}]$ that are initially located inside the cylinder shown in Fig. 3.9, would have crossed the surface dS a time dt later. The volume of such a cylinder could be evaluated as:

$$dV = dS\ v\ dt \cos\theta \tag{3.29}$$

So, the number of particles with velocities $\vec{v} \in [\vec{v}, \vec{v} + \vec{dv}]$ that are located inside the cylinder shown in Fig. 3.9 at a certain instant, would be given by the following expression:

$$dN(\vec{v}) = dV \ f(\vec{v})\vec{dv} \tag{3.30}$$

And, consequently, the number of particles with velocities $\vec{v} \in [\vec{v}, \vec{v} + \vec{dv}]$ that cross the xy plane per time and surface units, is given by:

$$\Phi(\vec{v}) \ \vec{dv} = \frac{dN(\vec{v})}{dS \ dt} = v \ \cos\theta f(\vec{v}) \ \vec{dv} \tag{3.31}$$

where, $v \cos\theta = v_z$, so:

$$\Phi(\vec{v}) \ \vec{dv} = v_z \ f(\vec{v}) \ \vec{dv} \tag{3.32}$$

As we are interested in the flux, through the xy plane and in the z direction, of particles with any velocity, we have to integrate Eq. (3.32) for every velocity $\vec{v}$ for which $v_z > 0$. That is:

$$\begin{aligned}\Phi_0 &= \int_{\forall \vec{v}, v_z > 0} \Phi(\vec{v}) \ \vec{dv} = \int_{-\infty}^{\infty} dv_x \int_{-\infty}^{\infty} dv_y \int_{0}^{\infty} dv_z \ v_z \ f(\vec{v}) \\ &= \underbrace{n \sqrt{\frac{kT}{2\pi m}} \exp\left(-\frac{m}{2kT} v_d^2\right)}_{\Phi_{\text{th}}} + \underbrace{n \ \frac{v_d}{2} \left(1 + \ \text{erf}\left(\sqrt{\frac{m}{2kT}} v_d\right)\right)}_{\Phi_{v_d}}\end{aligned} \tag{3.33}$$

The first term in the right hand side of Eq. (3.33), denoted as Φ_{th}, is the part of the flux due to the thermal motion of particles, while the second term, denoted as Φ_{v_d}, is the part of the flux due to their drift velocity. If we consider the case $v_d = 0$, we have that $\Phi_{v_d} = 0$ and $\Phi_{\text{th}} = n\sqrt{k_B T / 2\pi m}$, which is the thermal flux due to purely maxwellian particles through a surface. On the contrary, if we consider the case $k_B T = 0$, we have that $\Phi_{\text{th}} = 0$ and $\Phi_{v_d} = n v_d$, which is the flux due to monoenergetical particles moving with velocity v_d.

Now, by defining the thermal velocity as $v_{\text{th}} = \sqrt{2 k_B T / m}$ we can write Eq. (3.33) in the following way:

$$\Phi_0 \left(v_d, v_{\text{th}}\right) = \frac{n v_{th}}{2} \left[\frac{1}{\sqrt{\pi}} \ e^{-\left(\frac{v_d}{v_{th}}\right)^2} + \frac{v_d}{v_{th}} \left(1 + \ \text{erf}\left(\frac{v_d}{v_{th}}\right)\right) \right] \tag{3.34}$$

In Fig. 3.10, it is shown the dependence of this flux with respect to the ratio of the drift to thermal velocity. There, we can see how the two limiting cases of thermal and drift velocity driven flux, are recovered. Also, it can be seen that when $v_d / v_t h \gtrsim 1$ the flux can be approximated by the flux due to a monoenergetic beam of particles:

$$\Phi_0 \left(\frac{v_d}{v_{\text{th}}} \gtrsim 1\right) \sim n v_d \tag{3.35}$$

Despite the fact that we have obtained the approximation shown in Eq. (3.35), we will always use Eq. (3.34) to evaluate the number of particles that enter the simulation per time unit. However, the graph in Fig. 3.10 will become particularly interesting when analysing the results of the simulation of the cylindrical probe.

Finally, the velocity distribution of particles entering the simulation is given by $f_{\text{inc}}(\vec{v}) \propto v_z f(\vec{v})$. Where v_z is the component of the velocity normal to the influx surface, and $f(\vec{v})$ is given by Eq. (3.28).

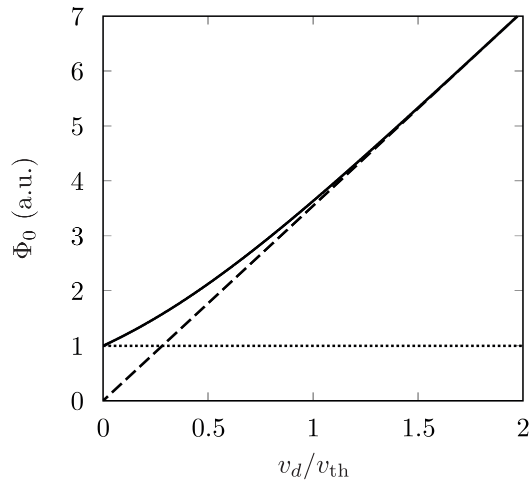


Figure 3.10: Transition from purely thermal to drift velocity driven flux. Solid line correspond to Eq. (3.34), while the dashed one correspond to $v_{\text{th}} = 0$ and the dotted one to $v_d = 0$.

The previous velocity distribution can be factorised in order to obtain the distribution for the different components of the velocity. By doing so, we obtain that the distribution for the component of the velocity normal to the surface is given by the following Rayleigh distribution:

$$f_{\text{inc.}}(v_z) = v_z\sqrt{\frac{m}{2\pi k_B T}}\exp\left(-\frac{m(v_z - v_d)^2}{2k_B T}\right) \tag{3.36}$$

while the distribution for the other components of the velocity is given by a Gaussian distribution:

$$f_{\text{inc.}}(v_x) = \sqrt{\frac{m}{2\pi k_B T}}\exp\left(-\frac{mv_x^2}{2k_B T}\right); \quad f_{\text{inc.}}(v_y) = \sqrt{\frac{m}{2\pi k_B T}}\exp\left(-\frac{mv_y^2}{2k_B T}\right) \tag{3.37}$$

We have to notice that, during the calculations performed in this section, without loss of generality, we have assumed a cartesian geometry where the drift velocity is assigned to the z direction. Nevertheless, as long as the drift velocity is perpendicular to the surface, all the calculations performed here are still valid.

We have already discussed most of the theory behind PIC simulations that is needed to develop our simulations. In the following section we will discuss the parallelisation techniques that we have used.

3.4. Need of parallelism and the GPGPU approach

Even though the basic theory behind PIC simulations, or in general any particle simulation, was already developed almost four decades ago, they have become relevant tools in the study of complex systems only in the past fifteen years approximately. This should come at no surprise.

On the one hand, obviously, performance of computer experiments is intimately related to the performance of computers themselves. And, it is well known that, raw performance of computers has grown exponentially since the early ages of computation. For example, the clock speed of processors has quickly increased from kHz, to MHz and from MHz to GHz. Also, as stated by the well known Moore's law, the number of transistor packed into a single microprocessor is doubled roughly every two years. These facts, among others, have allowed the exponential evolution in the number of operations that a computer can perform per second. Let us notice that, as can be seen in Table 3.2, the time needed by a processor to perform a single operation has been reduced from microseconds, in the early 70s, to nanoseconds, nowadays. This means that, computers are more than a thousand times faster now than forty years ago. So, because of the huge number of operations needed to perform just one iteration of a particle code, it has been only recently when computers have been powerful enough to run such simulations.

Processor	**Year**	**Clock frequency**	**#transistors**	**time per operation**
4004	1971	740 kHz	2300	1.35 μs
8080	1974	2 MHz	6000	0.50 μs
8086	1978	5 MHz	20000	0.20 μs
8088	1979	10 MHz	29000	0.10 μs
80286	1982	12 MHz	134000	83.3 ns
80386	1985	16 MHz	275000	62.5 ns
80486	1989	25 MHz	1180235	40.0 ns
i860	1991	50 MHz	N. A.	20.0 ns
Pentium	1994	100 MHz	3100000	10.0 ns
Pentium III	1999	600 MHz	9500000	1.67 ns
Pentium IV	2003	3.2 GHz	169M	0.31 ns
i7 980X	2010	3.3 GHz	1170M	0.30 ns
i7 5960X	2014	3.5 GHz	1400M	0.29 ns

Table 3.2: Raw performance of different Intel processors from the early 70s till nowadays. Number or transistors, and time per operation are approximate values.

On the other hand, even though performance of computers has been greatly improved, it is not enough to explain the recent wide spreading of particle simulations. There is another mayor breakthrough in computer technology that has allowed this growth, and this is parallelism. As can be seen in Table 3.2, during the last ten years the clock frequency of processors has not changed that much due to thermal restrictions. However, the density of transistors that can be packed into a processor is still increasing[2]. For this reasons, since ten years ago, the trend in the processor manufacturing industry has been to develop multicore processors. That is, processors with more than one ALU (Arithmetic Logic Unit), which can perform more than one instruction at a time.

To put it in perspective, let us do some approximate calculations. According to some rough estimations [50], the number of operations needed to do a single time step iteration is:

$$10N_p^2 - N_p \quad \longrightarrow \quad \text{PP model} \tag{3.38a}$$

$$20N_p + 5N_m^3 \log_2 N_m^3 \quad \longrightarrow \quad \text{PM model} \tag{3.38b}$$

N_p being the number of particles to simulate and N_m the number of nodes in any direction of a three dimensional mesh.

In Table 3.3 we can see the approximate number of operations needed to perform a single iteration of a particle simulation for both, PP and PM models. We have assumed typical values for the number of particles and mesh points, as we are interested in the order of magnitude only. Also, two times per operation, τ, have been assumed, for old and new hardware. As we can see, the execution times are prohibitive for PP and PM models executed in old hardware. On the contrary, for new hardware, the execution time for PP models are still prohibitive for large number of particles, and moderately assumable for PM models. For this reason particle simulations with reasonable numbers of particles have only be used in the recent past.

	PP model	**PM model**
#operations	10^{11}	2×10^8 s
execution time old hardware ($\tau = 10^{-6}$ s)	10^5 s	2×10^2 s
execution time new hardware ($\tau = 10^{-10}$ s)	10^1 s	2×10^{-2} s

Table 3.3: Time needed to perform sequentially all the operations of a single iteration. $N_p = 10^5$, $N_m = 128$.

We have to notice that, in the results shown in Table 3.3, we have considered only the time needed to perform the calculations, neglecting the time needed for any other task that needs to be done in any computer simulation. In particular, memory accesses have not been taken into account. As we will see in the next section, memory instructions can have a huge impact in the global performance of the code, since their characteristic time can be a few orders of magnitude higher than τ.

So, if we consider a reasonable number of iterations, $10^5 - 10^6$, and take account of the memory accesses, even PM models constitute and enormous challenge even for the most up to date hardware. However, it has to be said that, until now, we have considered a sequential execution of the simulation. That is, the computer perform one action at a time in a sequential fashion. Nevertheless, we have already mentioned that, in the last ten years, computing technology has move towards the use of multicore processors in order to spread the workload over more than one ALU. Obviously, if a certain number of processors, NP, are available, the time to execute a program can be divided almost[3] by the same number.

In order to take advantage of the execution across multiple processing units, a program must be divisible into different parts that are unrelated from each other, so that they can be executed independently at the same time. There are two main kinds of parallelism:

Task parallelism: when independent tasks are performed by different processing units.

Data parallelism: when the same task is performed by different processing units using different data.

[2]Manufacturers realised some time ago that they are about to reach the transistor density peak, and even Gordon E. Moore recently stated that its famous law will stop being valid in about ten years. This is due to the fact that, transistors are reaching the atomic size, and thus they can not be further miniaturised.

[3]Even in the case of a 100% parallel code, due to the time involved in communications between the different processing units, the execution time is never divided by the number of processors. This fact is known as Amdahl's law.

Particle simulations take advantage of the second kind of parallelism, *i. e.* data parallelism. As an example, we can think in the particle mover algorithm. Once the electric field, $\vec{E}_l^{\,p}$, suffered by each particle at a certain instant is known, Eqs. (3.27) have to be solved in order to obtain the new particle positions and velocities. If there are N_p particles in the system, this implies N_p equations sets that have to be solved, the resolution of which is independent from each other. So, the evaluation of Eqs. (3.27) for each $l \in [1, N_p]$ can be performed at the same time. Ideally, if we had N_p processors, each one would handle the evaluation of the new position and velocity for one single particle, so all the particles would be moved at the same time. Let us notice that this is data parallelism, since each processor is performing the exact same task (evaluate Eqs. (3.27)) but on different data sets (position, velocity and electric field of each particle).

By using the same example of the particle mover, we can see that, PIC simulations are extremely parallelizable. That is, PIC codes have a huge percentage of their operations that can be done in a parallel fashion. However, there is no code 100% parallel as there is always some kind of sequentiality. For example, by looking at Eqs. (3.27), we can see that Eq. (3.27b) has to be evaluated before Eq. (3.27a), since the latter depends on the result of the former. The remaining parts of PIC simulations are also highly parallelizable as we will see.

Now it is clear that our codes: on the one hand, perform a huge number of operations per time step, and thus, require huge computational resources, and on the other hand, most of the operations that have to be performed consist of the evaluation of the exact same expression with different data, and thus, can be performed in parallel (data parallelism). We only have to decide which technology to use in order to implement the parallelisation in our code.

There are multiple approaches when it comes to implement parallelism, being the main difference between them the hardware that is going to be used. Before the appearance of multicore processors, in order to execute parallel programs, multiple computers had to be connected, so the program can be executed in all of them. Typically, this was done with ethernet cables and using a communication library, such as MPI (Message Passing Interface), in order to coordinate the execution of the program in the different processors and to share data among them. This setup, which is usually called *cluster*, has been, and still is, one of the main approaches to parallelism. However, there are some flaws to this approach:

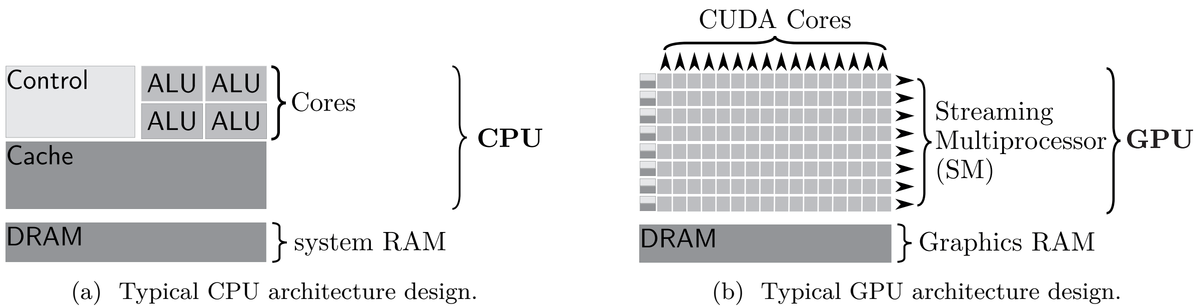
- In order to have a reasonable impact in the performance of the simulations, the number of nodes (this is how the different computers in a cluster are referred to) needed is high.
- Communications between different nodes are orders of magnitude slower than any other part of the code, even memory accesses, which already are orders of magnitude slower than operations in the processor.
- Cluster infraestructures are big and expensive. Modern clusters, even small ones, are complex systems which require: expensive hardware, a lot of power, a lot of space, room cooling systems, etc.
- Cluster management is not a trivial task, and require specific knowledge.

Most of the previous flaws, have been diminished since de advent of multicore processors. Modern CPUs (Central Processing Units) have multiple cores, and can be used as small but fast clusters. Small because the limited number of cores that a single CPU have. Even the most up to date, top of the line, server grade, CPUs available have a maximum of 16 physical cores (with technologies such as Intel's hyperthreading each core behaves virtually as 2). While modern clusters could have hundreds, if not thousands, of available cores. And fast because, as all the cores are within the same computer, external communications between nodes are avoided.

However, the most extreme multicore processors are found in GPUs (Graphic Processing Units). Because of the specificity of GPUs and the parallel nature of the algorithms that it typically executes, when it comes to multicore processor architecture, the approach of GPU manufacturers has been different compared to CPU manufacturers. The main difference between both architectures is shown in Fig. 3.11.

The total size of the processor is roughly the same in CPUs and GPUs, so, as we can see in Fig. 3.11, the main difference between both is how the space in the die is used, *i. e.* how many transistors are devoted to each task. CPUs devote less space to less, but big and powerful, cores and more space to big on-chip memory and control unit. While, on the other side, GPUs devote most of the space to a

massive number of small, and not so powerful, cores and less space to small on-chip memory and control units. We have to notice that in Fig. 3.11b we have used the CUDA terminology, more of which will be explained in the following section.



(a) Typical CPU architecture design.

(b) Typical GPU architecture design.

Figure 3.11: CPU vs GPU approach to multicore processors.

Previously, we stated that 32 is the maximum number of parallel tasks that a single CPU can perform, 16 cores $\times$ 2 threads per core. While a top of the class GPU, such as the Nvidia TITAN X have 3072 CUDA cores, grouped into 24 SM (Streaming Multiprocessors), each one capable of executing up to 2048 parallel threads, giving a maximum number of parallel threads of 49152 in a single GPU. We have to remember that, this kind of massively parallel processor, is obtained at the expense of less powerful cores and a small on-chip memory, when compared with a CPU. However, when it comes to execute massively parallel codes, such as the ones we are dealing with, it is difficult to beat the performance of a GPU without a large cluster. And, obviously, the GPU represents a simpler, cheaper, quieter and more power efficient approach.

Then, the last question to answer is, how to execute a simulation into a GPU. GPUs are obviously designed to handle graphics in a computer. Nevertheless, when developers realised the huge potential of GPUs as parallel processors, several APIs and programming languages appeared in order to use them as general processors. That was when the term GPGPU (General Purpose Graphics Processing Unit) was coined.

In the following section we are going to explain the main characteristics of the API and programming language that we chose to develop our simulations.

3.5. The CUDA® framework

In November 2006, Nvidia introduced CUDA® (Compute Unified Device Architecture), a general purpose parallel computing platform and programming model. CUDA is the API and programming language that our simulations have been developed with. It was designed to provide an easy way to develop general purpose software that takes advantage of the parallel capabilities of Nvidia GPUs.

The CUDA programming language is based in C/C++, which constitutes a high level language easy to learn for developers. It includes several extensions that allow the management of the GPU as well as the execution of code into it. However, CUDA supports other programming languages such as Fortran, Java, Python, etc. It also is included in several libraries which can be readily used without any knowledge about CUDA, *e. g.* in Mathematica and MATLAB.

At the core of the CUDA programming model there are three key abstractions: a hierarchy of thread groups, shared memories, and barrier synchronization. These abstractions provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. They guide the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel by blocks of threads, and each sub-problem into finer pieces that can be solved cooperatively in parallel by all the threads within the block.

By using this programming model, threads are allowed to cooperate when solving each sub-problem. Also, automatic scalability is enabled. When executing a CUDA program, each block of threads can be scheduled on any of the available SM within the GPU, in any order, concurrently or sequentially. So, a compiled CUDA program can be executed on any number of multiprocessors as illustrated in Fig. 3.12,

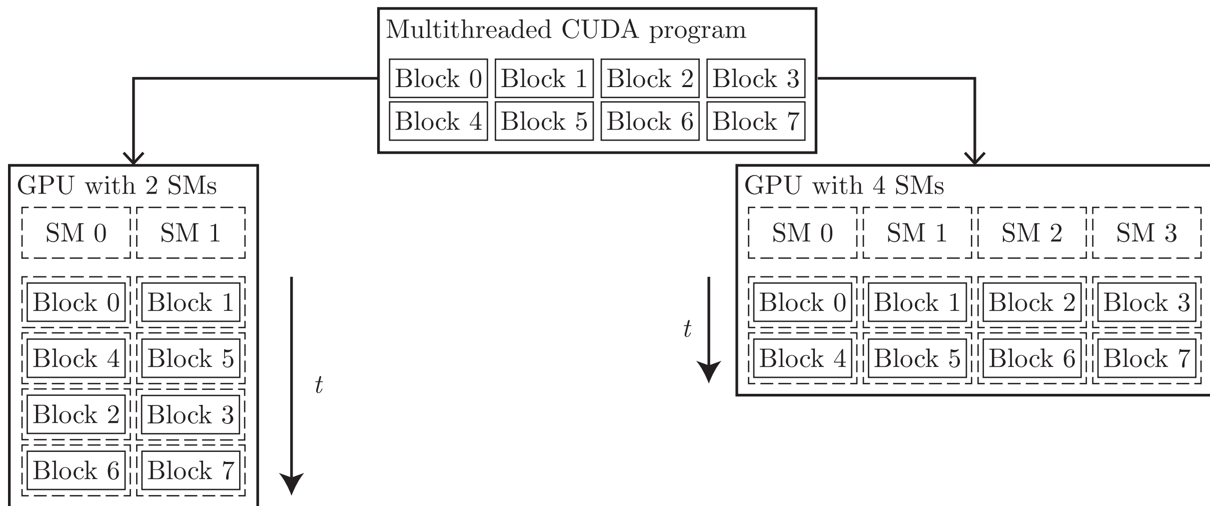


Figure 3.12: Automatic scalability enabled by the CUDA programming model.

and only the runtime system needs to know the physical multiprocessor count. This scalable programming model allows the GPU architecture to evolve between one generation and the following by simply scaling the number of multiprocessors and memory partitions.

For further details on the CUDA ecosystem, hardware implementation, current version, etc. the reader is referred to the online CUDA documentation provided by Nvidia [53]. However, in the following section we are going to introduce the basic concepts of the CUDA programming model.

3.5.1. Thread hierarchy

As we have previously introduced, the CUDA programming language is based in C/C++. Basically, CUDA C/C++ consist of a set of extensions that allow the parallel execution of pieces of code in a separate "device", which is the GPU, opposing to the "host" where the main code is executed, which is the CPU.

The aforementioned extension of the plain C/C++ provided by CUDA, is obtained by allowing the definition of a special kind of functions, called "kernels". When called, a kernel is executed in the device `N` times in parallel. A kernel is defined by using the **`__global__`** declaration specifier and, once defined, it can be called anywhere from the host specifying its corresponding "execution configuration". The execution configuration of a kernel, which is given by using the syntax `<<<GridDim,BlockDim>>>`, specifies how many parallel instances of it are to be executed in the device. Each "thread", *i. e.* individual instance of the kernel, that executes the kernel has a unique thread ID, that is accessible within the kernel through the built-in **`threadIdx`** variable.

```
1  // Kernel definition
2  __global__ void VecAdd(float* A, float* B, float* C)
3  {
4      int i = threadIdx.x;
5      C[i] = A[i] + B[i];
6  }
7
8  int main()
9  {
10     ...
11     // Kernel invocation with N threads
12     VecAdd<<<1, N>>>(A, B, C);
13     ...
14 }
```

Code 3.1: Simple kernel that adds two vectors `A` and `B` of size N (N≤1024) and stores the result into vector `C`.

In Code 3.1, the source code of a simple kernel that adds two vectors is shown. As we can see, the declaration of the `VecAdd` kernel is preceded with the **`__global__`** declaration specifier. Inside the definition of the kernel, the **`threadIdx`** variable is used, so that each thread performs one pair-wise addition. If the size of the vectors that are being added is `N` then, the same number of threads must execute the kernel. This is specified by the execution configuration, during the kernel call. In the previous example, `N` threads are launched grouped into one single block of threads.

It has to be noticed that, **`threadIdx`** is a three component vector. In this way, threads can be identified with the elements of a one-dimensional, two-dimensional or three-dimensional array. This is a convenient feature, since it provides a natural way to invoke computation across the elements of a domain such as segments (vectors), surfaces (matrix) or volumes (three-dimensional arrays).

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Code 3.2: Simple kernel that adds two matrixes `A` and `B` of size N×N (N≤32) and stores the result into matrix `C`.

In Code 3.2, the source code of a simple kernel that adds two matrixes is shown. The code is pretty much the same that the Code 3.1. However, in this case, threads are grouped into a two-dimensional block of threads. In this way, each thread has a two-index thread ID, so threads can be mapped into the elements of the matrixes.

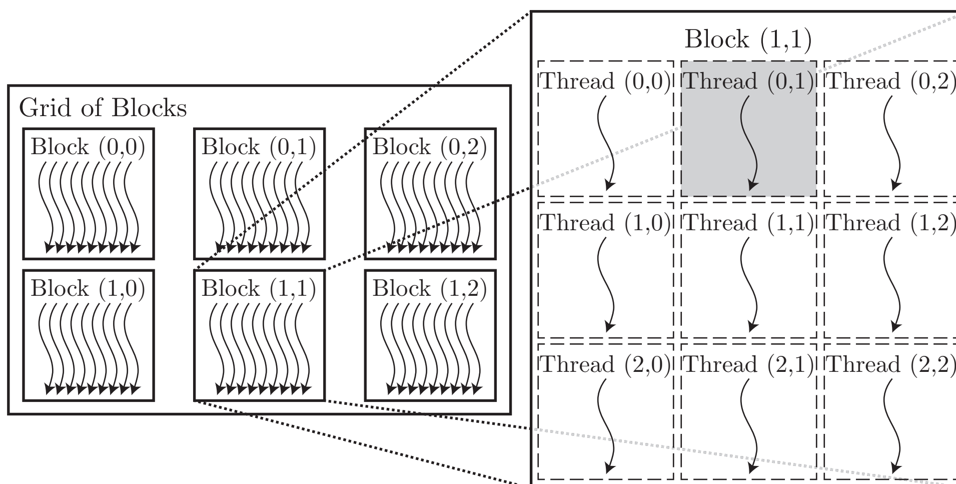


Figure 3.13: Thread hierarchy for a two-dimensional grid of two-dimensional thread blocks.

In the previously shown codes, a single block of threads executes the kernel (Line 12 in Code 3.1 and Lines 14 and 16 in Code 3.2), while the number of threads in the block depends on the size of the vector or matrix respectively, `N` or `N*N`. However, the number of threads that a block can contain is limited by

CUDA. Depending on the compute capability of the GPU that is being used, a maximum of 1024 threads per block are allowed. For example, in Code 3.2, if N is any larger than 32, the program would launch an error message at runtime and would stop its execution. Nevertheless, CUDA provides a feature to overcome such a problem. A kernel can be executed by a "grid" of equally shaped thread blocks. In the same way that threads can be grouped into 1D, 2D or 3D blocks of threads; blocks can be grouped into a 1D, 2D or 3D grid of blocks. In Fig. 3.13 thread hierarchy for a two-dimensional grid of blocks, as well as a two-dimensional block of threads, is shown. So, the total number of threads that execute a kernel is obtained as the number of threads per block multiplied by the number of blocks in the grid, which is unlimited.

Now, in order to unequivocally identify each thread, the block ID must be considered. In the same way that each thread within a block has a unique thread ID retrievable through the variable **threadIdx**, the ID of each block is retrievable through the variable **blockIdx**. For example, the highlighted thread in Fig. 3.13, has a value of **threadIdx** (0,1), and a value of **blockIdx** (1,1). Also, the dimensionality of a block can be obtained by each thread through the variable **blockDim**. The dimensionality of the grid must be specified in the execution configuration of any kernel call, just as the dimensionality of the blocks.

```
1  // Kernel definition
2  __global__ void MatAdd(float A[N][N], float B[N][N],
3  float C[N][N])
4  {
5      int i = blockIdx.x * blockDim.x + threadIdx.x;
6      int j = blockIdx.y * blockDim.y + threadIdx.y;
7      if (i < N && j < N)
8          C[i][j] = A[i][j] + B[i][j];
9  }
10
11 int main()
12 {
13     ...
14     // Kernel invocation
15     dim3 threadsPerBlock(16, 16);
16     dim3 numBlocks(N/threadsPerBlock.x+1, N/threadsPerBlock.y+1);
17     MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
18     ...
19 }
```

Code 3.3: Simple kernel that adds two matrixes A and B of size N×N (arbitrary N) and stores the result into matrix C.

A modified version of the Code 3.2, that is able to deal with matrixes larger than 32×32, is shown in Code 3.3. There, it can be seen that each thread block evaluates a submatrix of 16×16 elements (Lines 15 and 17 in Code 3.3). Also, a global thread ID is obtained by using the variables **threadIdx**, **blockIdx** and **blockDim** (Lines 5 and 6 in Code 3.3). This global index is used to map all the threads into the elements of the matrix, reason why the **if** statement is needed (Line 7 in Code 3.3).

3.5.2. Thread synchronisation and memory hierarchy

The different blocks within a grid, should be developed so that they could be executed independently from each other. This is a CUDA requirement, since they can be scheduled to be executed in any order, sequentially or in parallel, across any number of SM, as can be seen in Fig. 3.12. For this reason, threads are not allowed to cooperate across different block. However, they are allowed to cooperate with other threads within the same block.

In order to allow the cooperation between threads belonging to the same block, two features are offered by CUDA: synchronisation barriers and a shared memory space. Threads within a block can cooperate by sharing data, through a memory space which is accessible by all of them, and by synchronising their execution in order to coordinate the memory accesses.

The synchronisation between the threads that reside in the same block is performed with the intrinsic CUDA function **`__syncthreads`**`()`. This function, when called, sets a synchronisation point inside the code of a kernel, acting as a barrier at which the execution of all threads in the block is temporarily halted. Once all the threads of the block have reached the aforementioned synchronisation point, they can continue their execution.

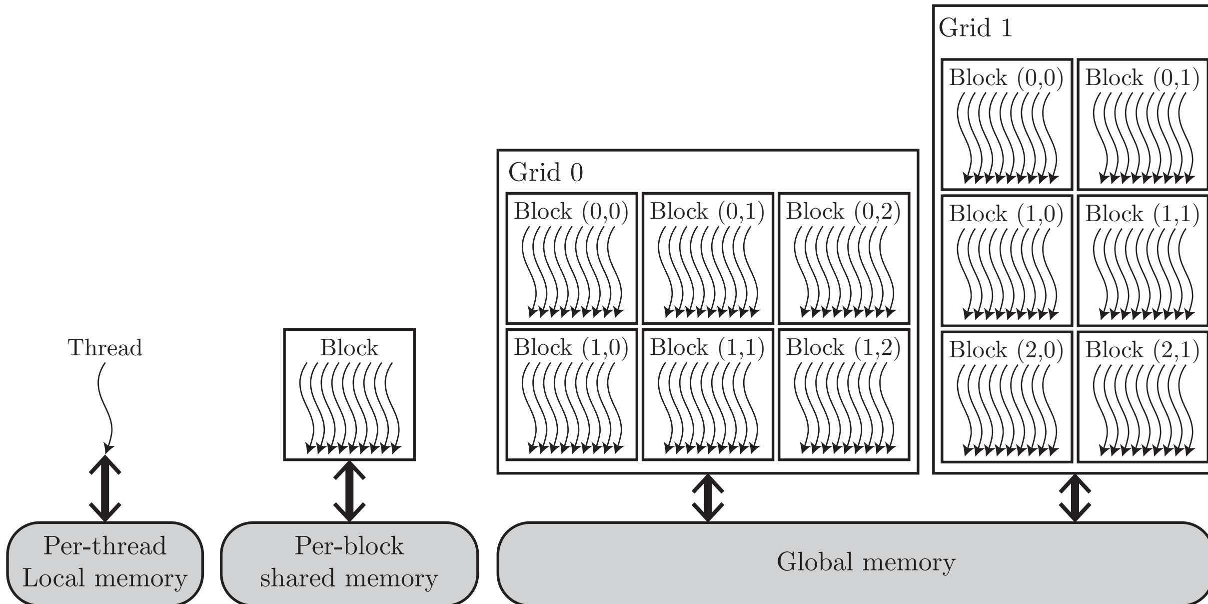


Figure 3.14: Memory hierarchy in a CUDA program.

In order for the cooperation between threads to be efficient, the memory space through which threads share data is expected to be low-latency and, the **`__syncthreads`**`()` function is expected to be lightweight. In order to accomplish the low-latency requirement for the memory, the memory hierarchy shown in Fig. 3.14 is available.

- The outer memory space is called "global memory", it is accessible by every thread of every block in every kernel launch. Global memory is large but very high-latency, much like system memory for the CPU, so it is not recommended to use it for cooperation between threads.
- Then, there is a shared memory space accessible only by the threads within a block and whose lifetime is the same of the block. This shared memory is low-latency and it is physically very near to each SM, similar to an L1 cache in a CPU. For this reasons, shared memory is intended to be used by the threads in a block in order to share data between them, as its name implies. However, the size of the shared memory space is limited, so its usage must be moderate.
- Last, but not least, each thread has its own local memory space, which is only accessible by it. This memory space shares most of the characteristics of the shared memory, but its size is even smaller. For this reason, special care must be taken in order not to misuse it. It is in this memory space where variables such as **`threadIdx`** are stored.

Additionally to the memory spaces shown in Fig. 3.14, that have been described previously, there are two more memory spaces: the constant and texture memory spaces. These read-only memory spaces are accessible by all threads, just as the global memory is. However, they are optimised for different memory usages and, since we do not used them in our simulations, we are not going to give further details about them.

3.5.3. Heterogeneous programming model

As we have mentioned before, the CUDA programming language consists of an extension to the C/C++ language that allows the execution of parallel code, *i. e.* kernels, in the device (GPU). However,

a CUDA program does not have to be composed exclusively of kernels, and so it can contain sequential code which is executed in the host (CPU).

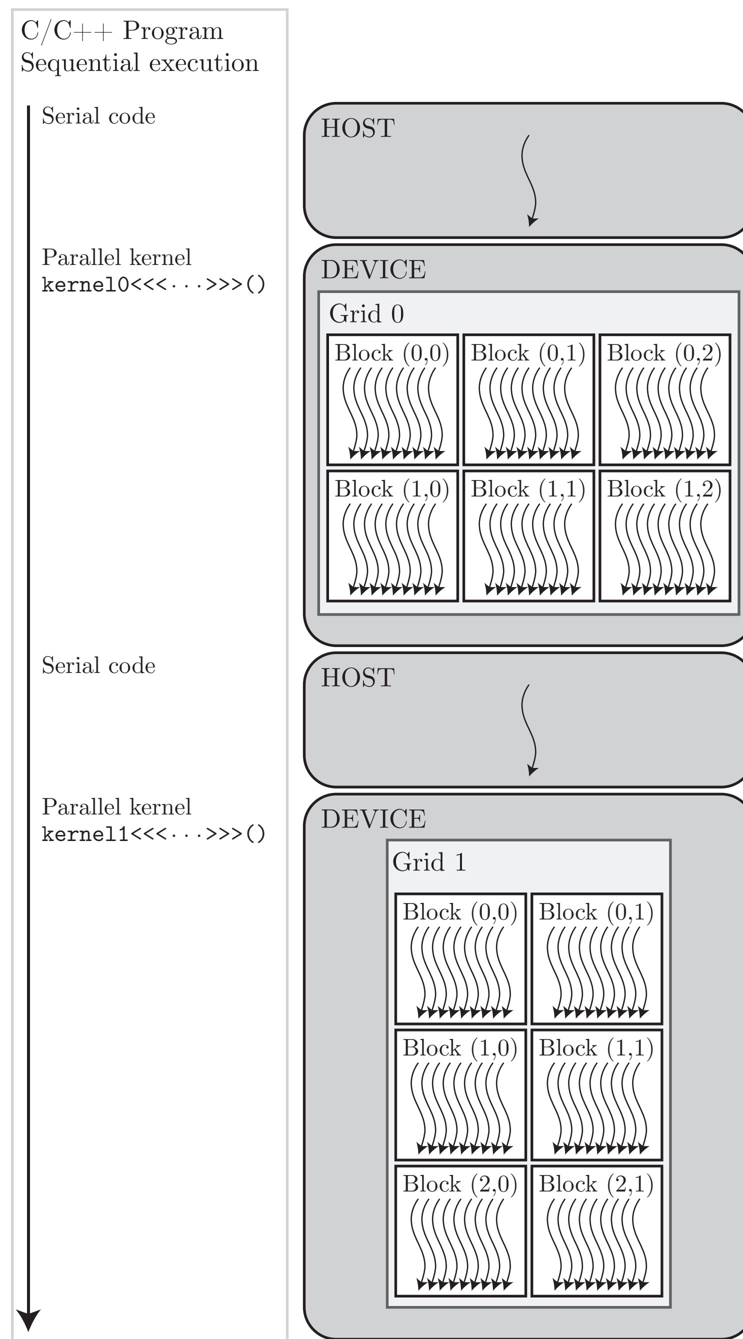


Figure 3.15: Scheme of a CUDA program showing the heterogeneous programming model.

In Fig. 3.15 the heterogeneous programming model of CUDA is shown by means of the scheme of a generic CUDA program. There we can see that, the C/C++ sequential code is executed in the host, just as in any regular C/C++ program. Then, when a kernel call is reached in the code, the execution control of the program is transferred to the device, which operates as a coprocessor to the host. After that, the device executes the kernel by a certain number of threads, grouped into blocks, which are scheduled in the available SMs of the GPU. Once the execution of the kernel is completed, the execution control of the program is transferred back to the host, until another kernel call is reached or the program finishes.

It has to be noticed that the host and the device are physically separated hardware. Accordingly, the CUDA programming model assumes that both, the host and the device, maintain their own memory spaces in DRAM. These two memory spaces are usually referred to as "host memory" and "device

memory". Actually, in the previous section, we have introduced the different kinds of memory available within the device memory.

Therefore, in order for the host and the device to cooperate, the CUDA programming language provides several functions for memory management. Among these, there are functions for allocation and deallocations of device memory, *e. g.* `cudaMalloc()` and `cudaFree()` , as well as functions to transfer data between host and device memories, *e. g.* `cudaMemcpy()`.

3.6. Conclusion

In this chapter we have given a general idea of the different techniques that can be used to simulate a plasma, paying special attention to particle models, since they overcome some of the problems that fluid models have. A brief introduction to particle models and their peculiarities have been made, and the computationally heavy nature of them has been highlighted. Then PIC models have been more thoroughly explained, since they are the models our simulations are based on. And, finally, the basic concepts behind GPGPU and CUDA, the parallelisation technique and framework that our simulations have been developed with, have been presented.

The following chapter is devoted to the description of the simulation of the contact of an infinite planar Langmuir probe with a plasma that we have developed, as well as the results obtained with it.

Chapter 4

PIC simulation of a planar Langmuir probe (CUPIC1D1V_PP)

4.1. Introduction

When studying the behaviour of electrostatic Langmuir probes immersed in plasmas, the infinite planar geometry is the simplest approach. Actually, it was the geometry considered in the first model that we developed of the contact of a plasma with a metallic surface, in Section 1.4. For this reason, even though one of the main objectives of this work is to study the behaviour of ions in the surroundings of a cylindrical Langmuir probe, first, we developed a simulation for the planar case.

As we will see, the simulation of the planar Langmuir probe will allow us:

- To develop the necessary skills, in PIC algorithms and CUDA programming, to establish a steady base from which to develop the cylindrical Langmuir probe simulation that we are interested in.
- To test the behaviour of the simulations, under simple conditions, by comparing their results with the ones provided by the fluid model developed in Section 1.4.
- To find the best initial conditions in order to quickly reach the steady state which is our interest.
- To develop a novel particle injection method that prevent the appearance of a "source sheath".

In this chapter, we are going to showcase the main characteristics and results obtained with the simulation that we have developed of the contact of a planar Langmuir probe with a plasma, `CUPIC1D1V_PP`. The complete source code of the simulation can be found in Appendix B. The meaning of the name given to the software can be explained as: "CUPIC" stands for CUDA PIC, "1D1V" remarks the dimensionality of the code[1] and, "PP" stands for Planar Probe.

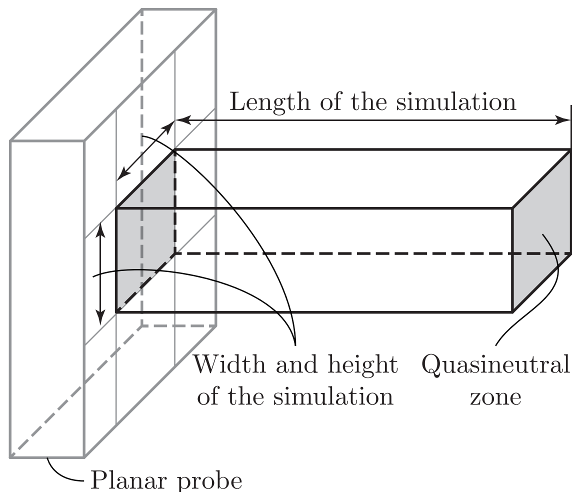
4.2. Computational abstraction of the system

In order to simulate any physical system, there are some steps that need to be taken to prior develop the actual code of the simulation. The reason being that, the real physical world and its magnitudes, have to be described in a way that results comprehensible by a computer.

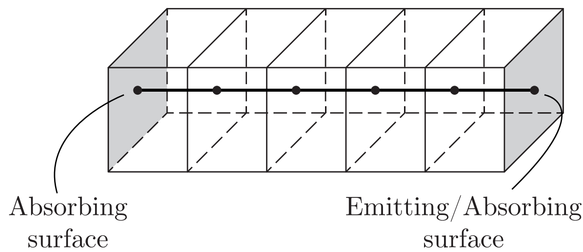
One of the first questions that needs to be addressed when developing a simulation is, what is the simulation domain? In other words, what is the physical space that we would like to simulate? In our case, the answer is clear, we want to simulate the space between a planar Langmuir probe and an unperturbed plasma. However, in order to avoid end corrections, we are considering the case of an

[1] It has to be noticed that, the dimensionality of PIC codes is usually expressed as nDmV, where n and m represent the dimensionality in ordinary and velocity spaces, respectively. In this sense, it is clear why the name of our simulation includes 1D1V, since we are only considering one dimension in both spatial and velocity spaces.

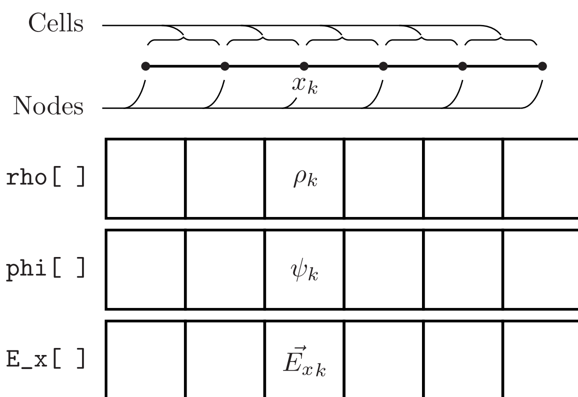
infinitely large probe, so the space between the probe surface and the plasma is also infinitely large. But, our computational box, and thus the simulation domain, has to remain finite in order to fit into the, also finite, computational resources available. Nevertheless, once the end corrections have been neglected, the only dimension that has to be considered is the one perpendicular to the probe surface. We have already used the same assumption in Chapter 1, when developing the fluid model of Section 1.4. In this way, the spatial dependence of all the physically meaningful quantities, is restricted to the dimension perpendicular to the probe surface.



(a) Simulation domain.



(b) Computational box and boundary conditions.



(c) Discretisation and computer variables.

Figure 4.1: Computational abstraction of the simulation domain between a planar Langmuir probe and a neutral plasma.

Taking into account the previous considerations, the simulation domain would be something like the one shown in Fig. 4.1a. There, we can see that the relevant dimension is the one perpendicular to the probe surface. The system we are going to simulate is the ensemble of particles inside the simulation domain of Fig. 4.1a. The width and height of the simulation domain are irrelevant, apart from the fact that they have to be adjusted in order to have a reasonable number of particles in the simulation. On the other hand, the simulation has to be long enough, so that the right hand side of it is located at some point along the preseath, where the quasineutral condition holds. We have to remember that, as we have developed a collisionless simulation, the quasineutral zone is infinitely long (see Fig. 1.10a).

In Fig. 4.1b, it can be seen an scheme of the computational box corresponding to the simulation domain shown in Fig. 4.1a. We have to notice that, since the simulation is unidimensional, the computational box is actually a segment. However, in Fig. 4.1b a three-dimensional abstraction is shown for the sake of ease of view. Also, the boundary conditions in the frontiers of the computational box are shown. The left hand side of it, represents the surface of the probe, so perfectly absorbing conditions are implemented there. At the right hand side, perfectly absorbing and also emitting conditions are implemented. This is because this frontier represents a certain point along the presheath, and so, particles enter and leave the simulation at this point.

Finally, the computational domain must be gridded in order to solve the field equations, *i. e.* Poisson's equation and field derivation. In Figs. 4.1b and 4.1c, the mesh of nodes, in which the computational box is decomposed, is shown. All the macroscopic magnitudes of the simulation are associated to this mesh, and the values of those macroscopic magnitudes at node positions are calculated every time step of the simulation. The values of each macroscopic quantity at node positions, are stored in the computer as an array of double precision floats, whose size is equal to the number of nodes in which the computational box is divided.

Once the simulation domain, as well as the macroscopic quantities associated with it, have been described, we have to think in the system that

is to be simulated. That is, the ensemble of particles that fill the space inside the simulation domain. As we did in the previous chapters when developing fluid models, we are going to consider a plasma composed of electrons and singly ionised ions only. So, in our simulation we have described two populations of particles: ions, and electrons.

Each particle in the simulation is described by the value of four attributes: its mass, its charge, its position and its velocity. It has to be noticed that, since our simulations is 1D1V, for each particle we are only considering one component of the position, *i. e.* the distance from the probe surface to the particle, and one component of the velocity, towards or backwards the probe.

In Fig. 4.2, an scheme representing the computational abstraction of the particle system is shown. Since there are two kind of particles present in our system, *i. e.* ions and electrons, we are considering two ensembles, one for each kind of particle.

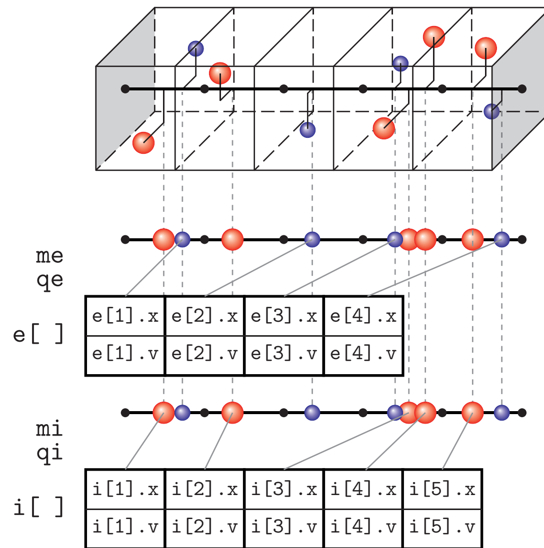


Figure 4.2: Computational abstraction of the particle system present in the simulation domain.

On the one hand, obviously, the values of two of the four attributes we are considering, are common for all the particles within the same ensemble, since the mass and the charge are defining attributes for each kind of particle. For this reason, we do not need to store the values of mass and charge for each individual particle in the system. Instead, we have four variables (doble precision floats) `me`, `qe`, `mi` and `qi`, that store the values of the electron mass, electron charge, ion mass and ion charge respectively.

On the other hand, in order to store the other two attributes that define each individual particle we used a `particle` structure, whose definition can be seen in Code B.14 (lines 29-33). There we can see that the particle structure has two members, both of which are double precision floats, to store the position and velocity of the corresponding particle.

As schematised in Fig. 4.2, each ensemble is represented by: two variables storing the charge and the mass of the corresponding type of particle and an array that stores the position and velocity of each particle in the ensemble. The elements of the array are of `particle` type, and its size is determined by the number of particles in the ensemble. It has to be noticed that, contrary to what it is shown in Fig. 4.2, particles are not sorted in the array. That is, the first element of the array does not have to store the position and velocity of the closest particle to the probe. Also, the size of the electron and ion arrays are adjusted dynamically as particles enter and leave the simulation.

Finally, once we have explained the computational abstraction of the domain that is going to be simulated, as well as the particle system that it contains, we have that the main variables in our simulation are:

- To characterise the mesh: `rho[]`, `phi[]` and `E_x[]`.
- To characterise the particles: `me`, `qe`, `e[]`, `mi`, `qi` and `i[]`.

Additionally there are many other variables, as can be seen in Appendix B where the source code of the simulation can be found. For example, variables that store the particles temperatures and drift velocities at the sheath edge, which will be commented in Section 4.3.5 when dealing with the plasma source. It also has to be noticed that, in the source code shown in Appendix B, variables usually appear preceded by `d_`, `h_`, `g_`, `sh_` or `reg_`, meaning that the variable is allocated in the device, host, global, shared or register memory spaces respectively.

Finally, there is one last thing that has to be stablished, the units that the simulation is going to work with. When performing any kind of numerical simulation in a computer, it is always desirable to work with numbers as close to the unity as possible. The reason is that, avoiding the use of huge or

tiny numbers, the rounding errors accumulated after millions and millions of arithmetic operations are diminished. The easiest way to work with numbers close to the unity, is by defining a set of dimensionless units where everything is measured in terms of the largest/smallest value of each magnitude. In our case, we have already defined such units. In Section 1.4, when developing a fluid model of the contact of a planar probe with a plasma, we defined a set of dimensionless units by Eqs. (1.21).

Once we have stablished the computational abstraction of the system that we want to simulate, we can explain the simulation itself.

4.3. CUPIC1D1V_PP implementation

In this section we will briefly discuss the parallelisation schemes and some other key aspect used in the main parts of the simulation. The explanations that we are going to give should provide the reader enough information to understand the complete sources of the simulation found in Appendix B. There, it is also provided a makefile that automates the compilation process, shown in Code B.17, as well as an example of the input file from which the simulation reads its parameters, shown in Code B.18.

4.3.1. Initial conditions and steady state

As we have previously mentioned, setting the initial conditions for our simulation is rather straightforward. We only have to specify an initial density and velocity distribution in the simulation domain. Once we have such information, the initial electron and ion ensembles are created. However, there are a few things related to the initial conditions of the simulations that should be highlighted.

It has to be noticed that, the steady state reached by our simulation should be independent of the initial conditions considered, depending only on the simulation parameters such as plasma density, particle temperatures, etc.. What really depends on the initial conditions is the transient state. During this state, the values of macroscopic quantities evolve from their initial values to their steady state values. Particularly, there exists a huge dependence of the transient duration upon the initial conditions.

For this reason, on the one hand, one of the first tests that we performed, in order to verify the correct behaviour of the developed simulation, was to verify the independence of the steady state on the initial conditions. To accomplish that, we executed two runs of the simulation with exactly the same parameters but different initial conditions. In case both runs produce the same results, it does not mean that our simulation is correct, but otherwise it would mean that the simulation is not working properly. On the other hand, we are interested in the steady state rather than the transient, which depending on the initial conditions considered would be more or less physically meaningful. So, in order to obtain results as fast as possible, we are interested in finding the initial conditions for which the steady state is reached the soonest.

To accomplish the previous objetives, the effect of different initial conditions on the simulation where studied. However, there are two main options when it comes to decide the initial conditions: to consider an empty system or to consider a system filled with particles. In this case, and for the sake of clarity we are going to showcase only two different initial conditions:

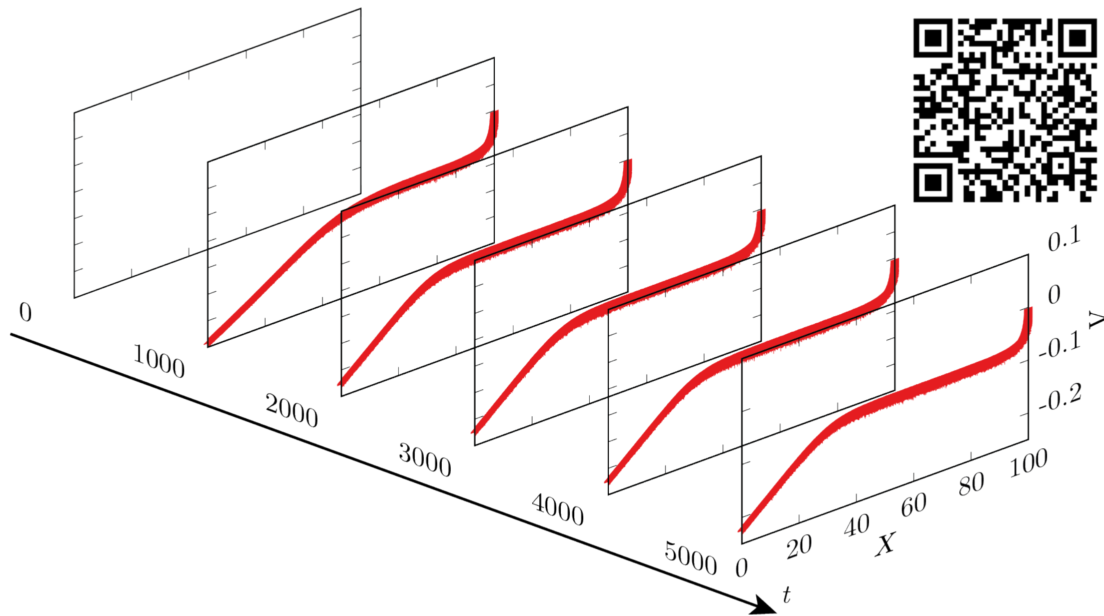
(a) The first initial conditions considered are very simple: the system is initialised completely empty.

(b) The second initial conditions considered are those corresponding to the plasma. Ions and electrons are thermalised, each one with its corresponding temperature, and their densities are equal and coinciding with their densities at the plasma.

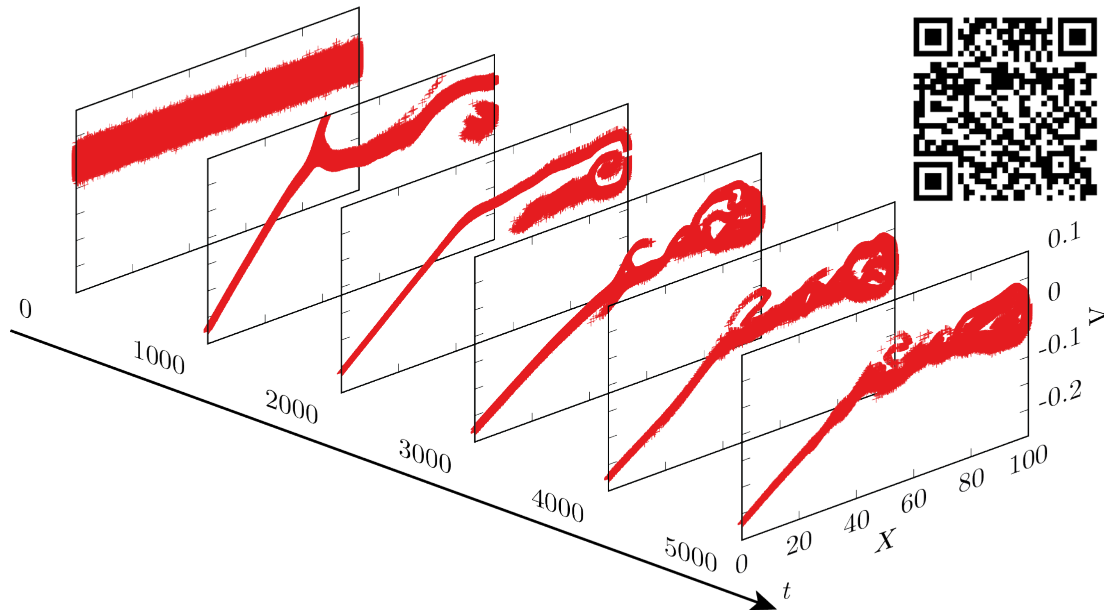
In Fig. 4.3 the dependence of the transient state on the different initial conditions is shown. This is done by means of representing the phase space of the ions ensemble at different time steps. On the one hand, Fig. 4.3a shows the evolution of the simulation with empty initial conditions. It can be seen that, as ions enter the simulation, they quickly adjust their properties to those of the steady state and, for $t \gtrsim 2500$, the system is already in its steady state configuration. On the other hand, Fig. 4.3b shows the evolution of the simulation whith initial conditions consisting of a filled system. There we can see that, even though the system is evolving towards the same configuration reached in Fig. 4.3a, the time needed

to reach it is much larger. As it can be seen, when a filled system is considered for the initial state of the simulation, the existing ions get trapped into potential wells in the right hand side of the simulation, *i. e.* $50 \leq x \leq 100$, the population of which slowly decreases. However, when initially there are no particles inside the computational box, the particles entering the simulation fall through a monotonic decreasing potential, and potential wells are not developed, thus trapped particles do not exist.

It has to be noticed that, the time-dependent nature of transient states makes difficult to show the evolution of system in an "static" document like the present one. For this reason, the reader is recommended to view the videos showing the evolution of the phase space by scanning the corresponding QR codes in Fig. 4.3 with a mobile device as, by doing so, the previously given explanations become much more clear.



(a) Initial conditions: empty system.



(b) Initial conditions: filled system.

Figure 4.3: Dependence of the evolution of the phase space for ions on the initial conditions. All magnitudes are expressed in simulation units.

4.3.2. Particle weighting

As we already stated in the previous chapter, the particle weighting algorithm is where the term PIC comes from. In this section we are going to review the implementation of this fundamental part of PIC simulations in our code. In particular, we will review the parallelisation strategy that we use and some aspects of its CUDA implementation, in order to explain concepts such as thread idling and atomic operations, which are used in the code of our simulation.

Let us start by remembering that, as we are going to use a CIC/PIC algorithm, the shape function that allows us to interpolate charge from the particle positions to the nodes is given by Eq. (3.17). By using this shape function, each particle contributes to the charge of its two nearest nodes, as can be seen in Fig. 3.7a. So, the algorithm implemented in our simulation can be resumed in the following steps:

1. As the nodes have fixed positions in the computational box, the two nearest nodes of each particle are found by looking their positions.

2. By using the shape function of Eq. (3.17), the corresponding fraction of the particle charge is added to each node.

Obviously, these rather simple steps must be done for every single particle in the system. In our case, we have two kind of particles: ions and electrons. However, in the rest of the section we are going to consider that we only have one kind of particle, since the algorithm is independent of the kind of particle. At the end we will see that, considering more that one kind of particle, consists simply in calling the function that perform the particle weighting with different arguments. In the case of serial execution, the code would be rather simple. The previous steps would be enclosed in a structure that loops over all the elements of the particle array. Nevertheless, the parallel execution is a little bit tricky, and thats what we are going to explain here.

Let us consider one kind of particle whose charge is q_p. In the computer the charge of those particles is stored in the variable `q`, and the particle ensemble itself is stored in an array `g_p[]` whose elements are of type `particle` and its length is stored in the variable `num_p` (number of particles). Also, the charge density at node positions is stored in the array `g_rho[]` whose elements are of type **`double`** and its length is stored in the variable `nn` (number of nodes). Finally, the spacing of our one-dimensional mesh is stored in the variable `ds`. With these variables as arguments, we define the kernel `particle_to_grid()` that performs the particle weighting in parallel. The complete definition of the kernel can be seen in Code B.4 (lines 145-205). We have to notice that the arguments of a kernel are stored in the global memory, reason why the names of some of the previous variables start with `g_`, in order to differentiate those variables from their corresponding versions in the shared memory.

First, we are going to explain the execution configuration of the kernel. If we have `num_p` particles, it is obvious that we want to launch an equal number of parallel threads, each one handling the weighting of a single particle. However, due to the restriction in the number of threads per block that can be requested (1024) and the large number of particles that it is usually considered (`num_p` $\gg$ 1024), the threads must be grouped into different blocks. In this sense, the execution configuration used in our simulation during the particle weighting is schematised in Fig. 4.4.

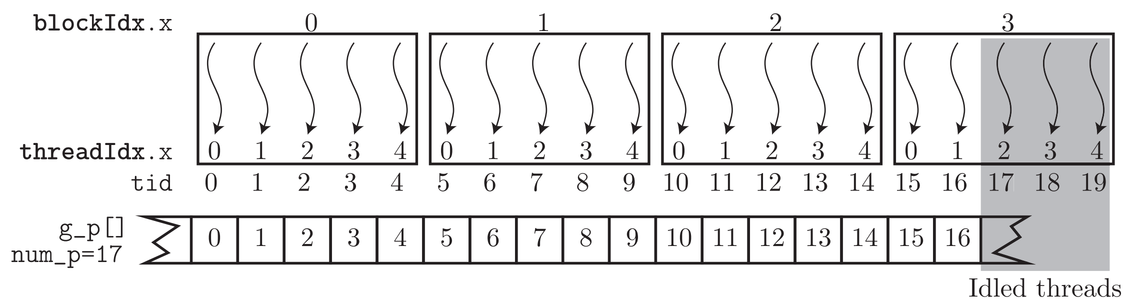


Figure 4.4: Scheme of the execution configuration of `particle_to_grid()` kernel.

The block size, *i. e.* the number of threads per block, that we choose for the `particle_to_grid()` kernel is 512, as can be seen in Code B.5 (line 21), while the number of blocks depends on the number

of particles that are in the simulation at the moment. In this way, each thread has a unique ID (Code B.4 line 153), tid, that allows it to know which particle to access within the array of particles in order to weight its charge into its two nearest nodes.

It has to be noticed that, in general, the total number of threads that are going to be launched is not going to be the same than the number of particles. Actually, the number of threads has to be larger or equal to the number of particles. But, all the threads within a block and all blocks within the grid must execute the same code (the kernel code). For example, if a thread has to read the memory address corresponding to the tid element of the g_p[] particle array, the thread number 2 of block number 3 in Fig. 4.4 would try to read g_p[17]. In this case, two things can happen: that the memory address is not allocated and the simulation halts throwing an error message, or, that the memory address is allocated but belongs to another array (*e. g.* another particle array) and thus a wrong datum is read without we even noticing it. Any of the previous options is not desirable in our code, so special care must be taken into account in order to idle the extra threads that should do nothing. This is done by evaluating the tid variable before any read or write operation, then, if its value is larger than num_p-1, the thread does nothing.

Thread idling is an important concept in CUDA programming and has to be thoughtfully considered when developing in CUDA. Specially because of the, sometimes obscure, debugging of wrong memory access errors in CUDA.

Once the execution configuration is established, when the kernel is called:

1. each thread reads the information of the tid particle,
2. each thread finds the two nearest nodes of particle tid and evaluates the contribution to their charge density,
3. each thread saves the contributions to the charge density due to particle tid in the corresponding elements of the array sh_rho[], which belongs to the shared memory of the block,
4. once all the threads within a block have finished to weight its corresponding particles, which is ensured by using the function **__syncthreads**(), the contributions of the particles analysed by each block are saved into the array g_rho[], which belongs to the global memory of the device.

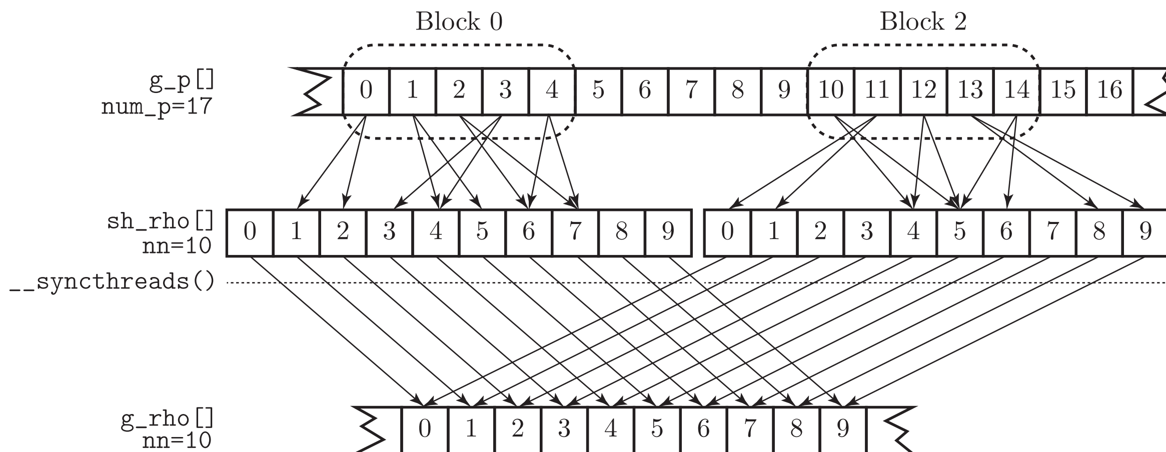


Figure 4.5: Write collisions during the execution of particle_to_grid() kernel.

In Fig. 4.5, the previous steps are schematised for blocks 0 and 2 of Fig. 4.4. Threads within a block usually perform the same tasks at exactly the same moment. For this reason, it is possible that different threads within a block attempt to write in the same shared memory address at the same time. Also, once all the threads within a block are finished, the information in sh_rho[] must be added to g_rho[] and, since different blocks can be scheduled to be executed at the same time, more than one block could try to write in the same global memory address at the same moment. This causes what is called a memory write collision. When write collisions are not properly handled the behaviour is, as stablished in the CUDA documentation [53], unpredictable. In order to properly perform the aforementioned memory writes, atomic functions must be used. The operations performed by such functions are atomic in the

sense that they are guaranteed to be performed without interference from other threads. In other words, no other thread can access these addresses until the operations are completed. This is the reason why `atomicAdd()` and `atomicSub()` functions were implemented, as can be seen in Code B.12 (lines 56-86). This functions allow multiple threads to add contributions to the charge density of the same node at the same time without interfering.

Although there are more details that need to be taken into account, a general idea of the implementation of the `particle_to_grid()` kernel has been given. The interested reader is referred to Appendix B, where its complete source code can be read. There, the definition of the function `charge_deposition()` can be found in Code B.4 (lines 15-55). This function handles the process of obtaining the complete charge density at node positions from the particles information. It makes two calls to the `particle_to_grid()` kernel, one for electrons and another for ions.

In the next section, we are going to explain how Poisson's equation is solved once that `g_rho[]` is obtained.

4.3.3. Poisson solver

The solution of Poisson's equation is at the core of the force evaluation method in a PIC simulation. In a parallelised simulation, where particles can be moved very quickly, as we will see in the next section, the solution of Poisson's equation constitutes one of the mayor bottlenecks of a PIC simulation. For this reason, it is probably the part of the code that needs to be optimised the most.

As we already stablished in Section 3.3.1, the approach that we have taken, in order to solve Poisson's equation, is the finite difference scheme. So, instead of solving an ordinary differential equation, we have to solve a system of linear equations. In our monodimensional case, the system that has to be solved is given by Eqs. (3.20a). It has to be noticed that, the number of equations and unknowns that we have to solve is given by the number of nodes in the mesh of our simulation. For this reason, we have opted for an iterative method to solve the system of equations, in particular the Jacobi method. Iterative methods are preferred for large systems, *i. e.* with more than 300 equations and unknowns, in order to diminish the numerical errors. In the method, we start with an approximation for the potential at node positions which is successively improved in terms of the previous approximations. In the Jacobi method, the new approximations are obtained by the expression:

$$\varphi_i^{\text{new}} = \varphi_i^{\text{old}} + \frac{1}{2}\left(h^2\rho_i + \varphi_{i-1}^{\text{old}} + \varphi_{i+1}^{\text{old}}\right); \quad \forall i = 1, \dots, N_n - 2 \tag{4.1}$$

h being the mesh spacing and N_n the number of nodes in the mesh. We have to notice that, in our case, there are two equations and unknowns less than mesh nodes, since the first and last nodes have a fixed value for the potential because of the boundary conditions. The iterative method stops when the error in one iteration is smaller than a certain threshold, and the error is evaluated as the maximum difference between the new and the old approximations in a node.

Now, we are going to explain the parallelisation scheme that we use in the implementation of the Jacobi method for the solution of Poisson's equation. We, will see how the use of shared memories and synchronisation barriers allow the cooperation between threads within a block. But first, let us establish the main steps that need to be performed in every single iteration of the Jacobi method:

1. obtain the new values of the potential in every node of the mesh,
2. obtain the maximum difference between the new and the old approximation of the potential across all the nodes of the mesh.

We have to notice that, the obtention of the new approximations of the potential and the difference between the new and old approximations for each node (node error) is a task 100% parallelizable. However, in order obtain the global error of the iteration, all node errors have to be compared, which is a task not so parallelizable and that implies thread cooperation.

So, in order to solve Poisson's equation in our simulation, we have developed the kernel called `jacobi_iteration()`, which performs one iteration of the method and evaluates the error in that iteration. Then this error is compared with a certain threshold and it is decided if another call to

jacobi_iteration() is needed. The definition of the kernel can be seen in Code B.4 (lines 207-267). It seem obvious that, for the execution configuration of this kernel, we launch as many threads as nodes in our mesh, nn, as by doing so each thread deals with the obtention of the new approximation and the error for one single node. However, since the values of the potential at the first and last nodes are fixed by the boundary conditions, as we will see in Section **??**, we only have to solve Poisson's equation in the inner nodes of the mesh. So, the number of threads that needs to be launched is nn-2. Because of the same reasons argued in the previous section, instead of specifying one single block with nn-2 threads in it, we divide the total number of threads into several blocks.

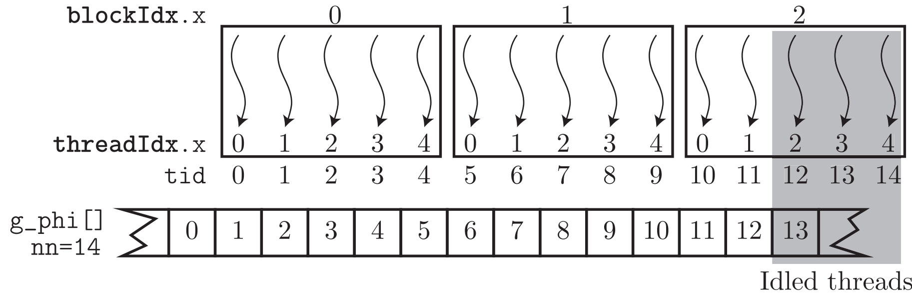


Figure 4.6: Scheme of the execution configuration of jacobi_iteration() kernel.

In Fig. 4.6 the execution configuration of jacobi_iteration() is schematised. As we can see, the configuration is almost the same as the one corresponding to the particle_to_grid() kernel (Fig. 4.4), however instead of a particle array now we are dealing with a mesh array. Also, for the same reasons as in the particle_to_grid() kernel, here we have to deal with idled threads. The main difference in the execution configuration is that, in the case of the jacobi_iteration() kernel, the number of threads per block is 128, as can be seen in Code B.5 (line 22).

Once we know the execution configuration, we are going to explain the steps taken within the jacobi_iteration() kernel:

1. each thread loads the value of the potential at its corresponding node position into the shared memory,
2. the first and last threads of each block, load the value of the potential at the previous and next nodes of the block,
3. each thread obtains the new approximations of the potential (saved into local registers) and the errors associated to its corresponding node (saved into the shared memory),
4. threads within each block cooperate to obtain the maximum error of the nodes corresponding to the block,
5. the first thread of each block saves its partial maximum error in the global memory.

In Fig. 4.7 an scheme of the previous steps can be seen. In the first two steps, all the values of the potential that are needed by the threads within a block are loaded into the shared memory. This allows each thread to read the three values of the potential needed to evaluate Eq. (4.1), from the shared memory (fast), instead of doing it from the global memory (slow). This constitutes the first example of thread cooperation. Instead of reading three values from the global memory, which is a very slow operation, each thread reads only one (except for two threads per block that read two values). Then, all the threads share the data they have read from the global memory through the shared memory. It has to be noted that, since in the third step the task performed by each thread relies on information provided by other threads, a synchronisation barrier has to be stablished after the second step. This ensures that the third step is not performed by any thread until all the threads within the same block have finished the second, thus all the required values of the potential are available in the shared memory. The third step is rather simple: each thread evaluates the new approximation of the potential in the corresponding node. Also, with the new and the old approximations, each thread evaluates the local error and saves it in an array in the shared memory. In the forth step we can see another example of thread cooperation. We can see how this step consist of a series of n iterations where 2^n threads are idled in each iteration

and, the rest of the threads compare two errors each and store the maximum in its corresponding shared memory address. After this, we end up with the maximum error of the block saved in the first element of sh_error[]. Finally, the fifth step consist of saving the partial maximum errors into the global memory, where the host can perform the final comparison to obtain the maximum error and decide if another iteration of the Jacobi method is needed.

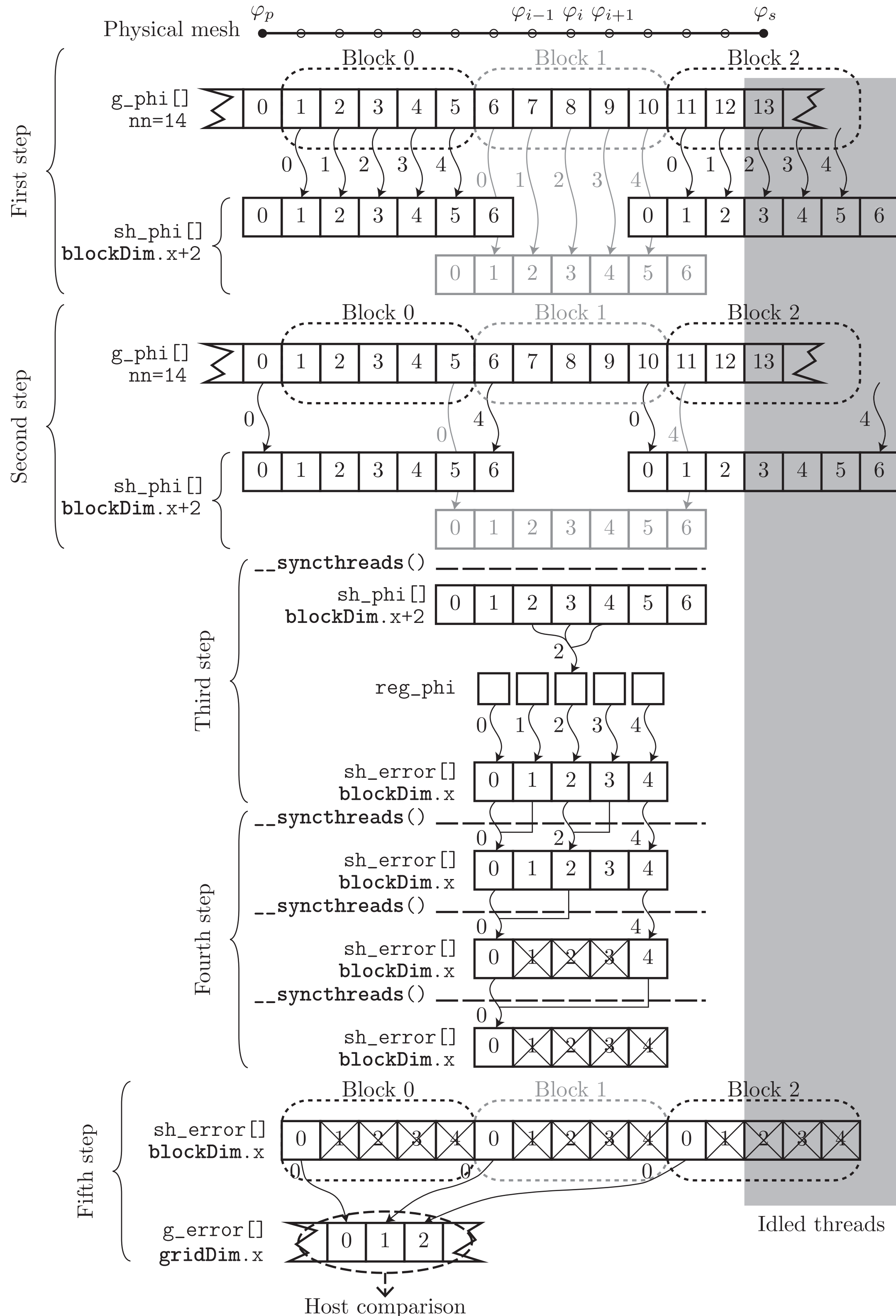


Figure 4.7: Scheme of the jacobi_iteration() kernel.

After the Poisson's equation is solved, the first derivative of the potential must be evaluated in order to obtain the electric field. This is done by using Eq. (3.22a) for each node. This task is performed in parallel by the `field_derivation()` kernel, whose definition can be seen in Code B.4 (lines 269-318). For the sake of briefness we are not going to cover here its implementation since, once the `jacobi_iteration()` kernel is explained, it is easy to understand it by reading the sources.

4.3.4. Particle mover

The particle mover is a fundamental part of any PIC simulation. As we explained in Section 3.3.2, the algorithm we are using to integrate the equations of motion for particles is the leap-frog algorithm. There, we stated that the positions and velocities of particles were updated by using Eqs. (3.27), which in our unidimensional case can be rewritten as:

$$v_l^{\,p+1/2} = v_l^{\,p-1/2} + \Delta t\ \frac{q_l}{m_l} E_l^{\,p} \tag{4.2a}$$

$$r_l^{\,p+1} = r_l^{\,p} + \Delta t\ v_l^{\,p+1/2} \tag{4.2b}$$

where the l index is referred to the particle in the system and the p index is referred to the iteration of the PIC algorithm.

It has to be noticed that, in order to solve Eqs. (4.2), we only need the particle attributes q_l and m_l, and the value of the electric field at particle position, $E_l^{\,p}$. However, since we know the value of the electric field at node positions, we have to perform the field weighting step that we talked about in the previous chapter.

In order to evaluate Eqs. (4.2), for each particle in the simulation, we define the kernel function `leap_frog_step()`, whose definition can be seen in Code B.6 (lines 64-114). Even though the name of the kernel is due to the algorithm used to integrate the equations of motion, in order to increase the computational load of the kernel, it also performs the field weighting step. This leads to a higher performance, since a higher operation to memory access ratio is achieved, and allows us to avoid the use of an intermediate array to store the values of the electric field at particle positions.

The execution configuration of both `leap_frog_step()` and `particle_to_grid()` kernels are exactly the same. This is explained because of their similarities, since both have to deal with particles and mesh arrays at the same time. The execution configuration can be seen schematised in Fig. 4.4, there we see that enough threads are launched so that each thread actualise the position and velocity of one particle only. Those threads are grouped into blocks of 512 threads each, as can be seen in Code B.7 (line 22).

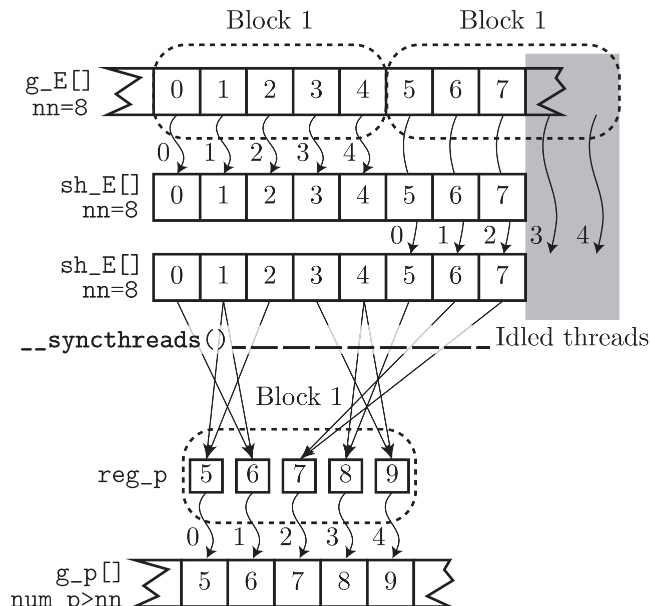


Figure 4.8: Scheme of the `leap_frog_step()` kernel. Only on block of threads is shown.

In Fig. 4.8 the workflow of each thread block that executes the `leap_frog_step()` kernel is schematised. First, the array that contains the values of the electric field is loaded into the shared memory. In order to perform this step, in case the number of threads per block is smaller than the number of nodes, several read operations are performed by each thread. Also, in general, several threads should be idled. Then, once the load of `g_E[]` into `sh_E[]` is completed, which is ensured by the use of a synchronisation barrier, each thread handles the actualisation of the position and velocity of one particle. First, each thread loads the information of its corresponding particle into the thread registers. Then, the two nodes from which the value of the electric field should be interpolated are found and, the electric field at the particle position is obtained. Finally, once the electric field at the particle position is

known, the thread evaluates Eqs. (4.2) in order to obtain the new values of position and velocity for its particle. Those values are then stored in the corresponding address in the global memory.

It should be noticed that, for the field weighting, we use the same shape function than the one used in the particle weighting step, in order to preserve momentum in our simulation. Also, as we stated in Section 4.3.2, in general, the number of threads that are launched is not going to be the same than the number of particles to move. For this reason, some threads of the last block should be idled, as shown in Fig. 4.4.

4.3.5. Particle injection and boundary effects

This is the last part of the code of our simulation we are going to discuss and, probably, the most interesting one from a physical point of view. It may seem that boundary conditions are straightforward to establish, but the truth is that the physical behaviour of the simulation greatly depends on small details in the boundary conditions configuration.

We have to notice that, in our simulation, two "sets" of boundary conditions must be stablished: one for the mesh, *i. e.* boundary conditions for the potential in order to solve Poisson's equation, and another for the particles, *i. e.* absorbing and emitting conditions for particles. Both sets consist of conditions established at the limits of our simulation domain/computational box. Since our simulation is unidimensional, we have two boundaries where conditions can be established. These boundaries can be seen greyed out in Figs. 4.1a and 4.1b.

On the one hand, boundary conditions at the left hand side of the simulation, *i. e.* the probe surface, are rather simple:

- The boundary condition for the potential at the probe surface consists in fixing the value of the potential in the first node of the mesh. The value at which `phi[0]` is fixed corresponds with the biasing potential of the probe, φ_p that we wish to simulate. As we have already mentioned in the previous section, this is the reason why the first element of the potential array is not changed during the execution of the Poisson solver.
- The boundary condition for the particles, since we are simulating a probe which is perfectly absorbing, consist in removing from the simulation all the particles that cross this boundary. Noticing that positions are measured as the distance to the probe surface, all particles with negative positions are immediately withdrawn from the corresponding particle array.

On the other hand, boundary conditions at the right hand side of the simulation are not that straightforward. One of the reasons for such a difference is that particles enter the simulation at this point, so emitting conditions are also needed. However, the main problem when defining the boundary conditions at this point is that, as we already introduced in Section 4.2, the right hand side of the simulation corresponds with a point along the presheath and not the plasma. As we are not taking into account ionisation in our simulation, the length of the presheath is infinitely large. So, no matter how long we made our simulation, its right hand side is going to be located along the presheath.

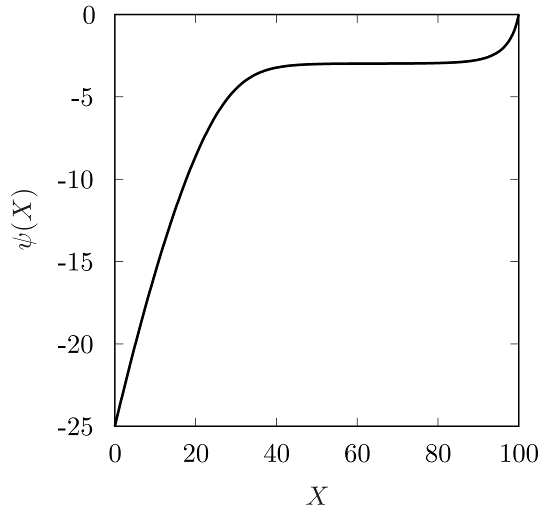
If we could impose plasma conditions to the right hand side of the simulation, its boundary conditions would be almost as simple as the left hand side ones. First, the potential of the last node of the simulation would be fixed with a value equal to the plasma potential. Since we are considering the plasma as the reference for the potential, this value would be zero. Then, as particles in the plasma can be assumed to be at thermal equilibrium, the influx of particles could be evaluated as the effusion of particles with a Maxwellian distribution. That is, the influx rate of particles can be obtained from Eq. (3.34) by considering $v_d = 0$, and the velocity distribution of the incoming particles would be given by the corresponding Rayleigh distribution.

We are going to see how the use of such boundary conditions lead to wrong results. Actually, these are the boundary conditions that we used in Section 4.2 to produce the results shown in Fig. 4.3. As it can be seen by observing the steady state reached in Fig. 4.3a, when using this boundary conditions, ions are accelerated near the right hand side of the simulation, then their velocity remains stable and finally they are accelerated again before reaching the probe. It has to be noticed that this is a quite strange behaviour compared with all that we know from fluid models. As we saw in Section 1.4, ions coming from the plasma should be slowly accelerated along the presheath until they reach the Bohm velocity, so

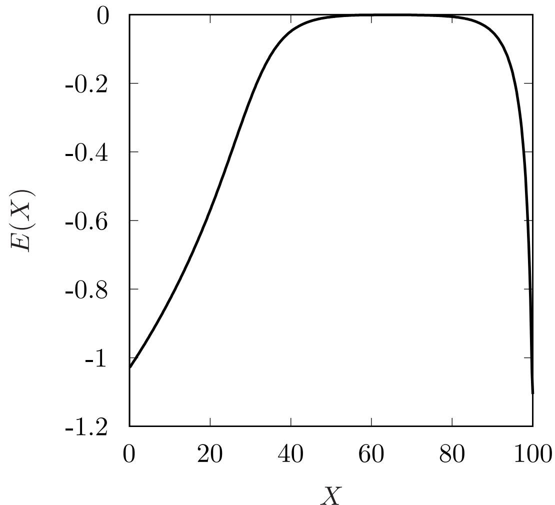
the sheath can be developed, and then ions are further accelerated along the sheath until they reach the probe.

In Fig. 4.9, the electric potential and field distribution corresponding to the aforementioned steady state is shown. In particular, this graph is obtained by means of averaging the data from iteration 400000 to iteration 500000. The strange behaviour of ions is explained once we see the potential distribution shown in Fig. 4.9a. There, we can see that the first acceleration of ions that takes place near the source of particles, is explained because of the potential drop occurring there.

Being the problem that, the potential distribution shown in Fig. 4.9a does not make any sense from a physical point of view. We have tried to impose boundary conditions in the right hand side of the simulation corresponding to those of the unperturbed plasma. Nevertheless, as it is readily seen in Fig. 4.9b, the value of the electric field at the right hand side of the simulation is far from being zero, which is the value it should have at the plasma. The reason is that, since we are not considering ionisation we can not try to impose plasma conditions at any finite distance from the probe, as we have previously said. The structure developed at the right hand side in Fig. 4.9 is usually called a “source sheath”, since is a sheath like structure due to the particle source. Source sheats are simulation artefacts due to the fact that ions are not properly injected into the simulation.



(a) Potential distribution



(b) Field distribution

Figure 4.9: Appearance of a source sheath during the simulation of a planar probe without a proper particle injection.

Then, the question is, how do we set up proper particle injection and boundary conditions for the potential at the right hand side of the simulation? The answer is found by understanding the nature of the presheath or quasineutral region that connects the sheath with the plasma. As we saw in Section 1.4, in order for the presheath to be able to connect the plasma with the sheath, ions are accelerated while maintaining quasineutrality, until they reach the Bohm velocity, so that the sheath can be developed. However, due to the continuity equation, when ions are accelerated their density tends to decrease. For this reason, a “presheath mechanism” that increases the ion density is needed, in order to partially compensate the decrease due to their acceleration, so that the ion density does not fall below the electron one. Therefore, the net effect of the presheath is to increase the flux of ions while maintaining quasineutrality.

Once it is understood what happens along the presheath, the conditions that we should impose in the right hand side of our simulation become clearer. On the one hand, we know that ions should have a certain drift velocity, which is acquired from the plasma to the point of the presheath where the right hand side of the simulation is located, which we will call the sheath edge. This drift velocity is due to a certain potential drop between the plasma, which is our reference for the potential, and the sheath edge. Also, because of the energy conservation, the drift velocity of ions and the potential at the sheath edge must fulfil the equation:

$$\frac{1}{2}m_i v_d^2 = e\varphi_s \Rightarrow \varphi_s = \frac{m_i v_d^2}{2e} \tag{4.3}$$

So, ions should be injected by considering that, at the sheath edge, their velocity distribution function is given by Eq. (3.28), so the influx rate is given by Eq. (3.34) and the velocity distribution of the incoming

particles is given by the corresponding Rayleigh distribution. On the other hand, as we are considering negative probe biasing potentials with respect to the plasma, electrons are repelled by the probe and thus their drift velocity should be zero. So, the expressions considered for electrons are the same but with $v_d = 0$. Obviously, because of the quasineutrality condition the distribution functions for ions and electrons should be normalised to the same particle density.

Then, the only remaining question is, what value should we use for the drift velocity? The answer to this question is not easy, since it depends on the length of the simulation, *i. e.* how far do we go into the presheath. Obviously, as the simulation is made larger this drift velocity is decreased, since it should be null at the plasma. However, we know that, as the quasineutral condition must hold at any point along the presheath, the electric field should be negligible there. So, the drift velocity for ions can be selfconsistently adjusted so that the electric field at the right hand side of the simulation becomes negligible. Also, as the drift velocity changes, the boundary condition for the potential at the right hand side of the simulation must be updated by using Eq. (4.3). This calibration of the ion drift velocity is performed by the function `calibrate_ion_flux()`, whose definition can be seen in Code B.8 (lines 219-254).

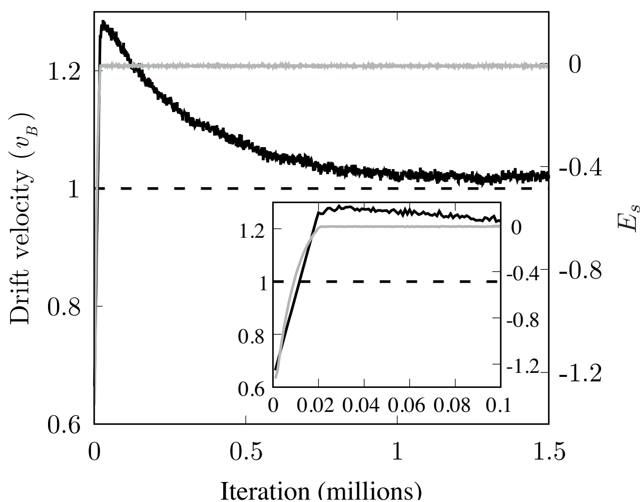


Figure 4.10: Calibration of the ion flux to avoid the appearance of a source sheath. Black solid line represents drift velocity and grey solid line represents the electric field at the sheath edge.

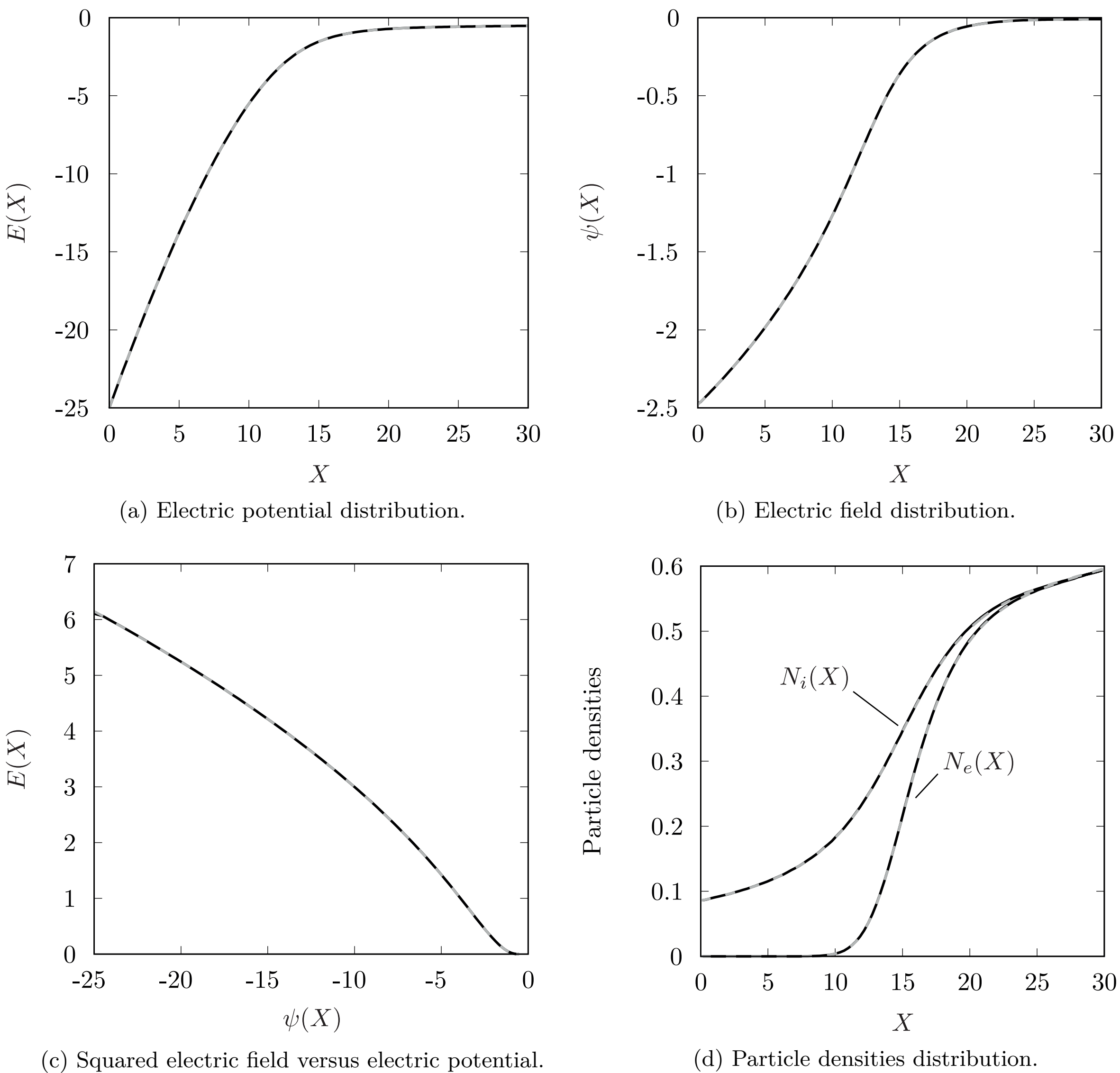
With the previously described method, as the simulation evolves, the electric field at the sheath edge slowly decreases. So, the source sheath that appears in Fig. 4.9 becomes smaller and smaller until it almost disappears. In Fig. 4.10 the evolution of the drift velocity for ions is shown, along with the evolution of the value of the electric field at the sheath edge, *i. e.* the right hand side node of the simulation. In order to compare the results with the simple model developed in Section 1.4, the temperature of ions was set to zero, so ions entering the simulation consist of a monoenergetic beam with velocity v_d. As it can be seen in the inset of the graph, the drift velocity is increased until the source sheath disappears, *i. e.* the electric field at the sheath edge becomes negligible. Then, after a transient stage, the drift velocity decreases to reach its stable value.

A few things have to be noticed. First, in Fig. 4.10 the drift velocity is measured in units of the Bohm velocity. Let us remember that, the Bohm velocity is the minimum velocity that ions should have in order for the sheath to be developed. So, as can be seen in Fig. 4.10, the stable value of the drift velocity is slightly higher than that, which implies that the calibration algorithm works flawlessly. Then, the reason why the drift velocity does not reach a stable value exactly equal to the Bohm velocity is that, the value we have chosen for the electric field at the sheath edge is negligible, but not zero. The actual value we have chosen is -10^{-2} in simulation units, as can be seen in Code B.8 (line 227). It is important to choose a slightly negative value, for the electric field at the sheath edge, because an electric field at the sheath edge exactly equal to zero would be compatible with any drift velocity higher than v_B. So, by doing this, the drift velocity would increase indefinitely, which is not a desirable behaviour of the simulation. The closer to zero that we choose the value of the electric field at the sheath edge, the closer to Bohm velocity that the stable value of the drift velocity is. However, because of the noise in the simulation, this value can not be chosen as small as we want, since it would become indistinguishable from zero, which would cause, as we said, the indefinite increase of the drift velocity.

4.4. Comparison with fluid models

Since we are going to use our simulations to study situations that are not covered by fluid models, it seems reasonable to see if it produces the same results that fluids model under simple circumstances. In order to do that, we have compared the results of the simulation shown in Fig. 4.10 with those provided by the fluid model developed in Section 1.4. In particular, we are going to compare the results of CUPIC1D1V_PP with the sheath solution because, as we have already stated, our simulation does not take into account ionisation processes.

On the one hand, the results of the simulation are obtained by means of averaging the values of the macroscopic quantities associated with the simulation, over a certain number of iterations. Obviously, in order to start averaging data, we have to be sure that the simulation has reached the steady state. In this case, we have averaged the data from iteration 1400000 to iteration 1500000, *i. e.* over half a million of iterations of the PIC simulation. On the other hand, the results of the fluid model are obtained in the same fashion that the ones shown in Fig. 1.9, but considering the same parameters as in the PIC simulation, *e. g.* same biasing potential for the probe.



(a) Electric potential distribution.

(b) Electric field distribution.

(c) Squared electric field versus electric potential.

(d) Particle densities distribution.

Figure 4.11: Comparison between PIC simulation and fluid model of Section 1.4. Black solid lines correspond to PIC simulation while grey dashed lines correspond to fluid model.

The previously mentioned results can be seen compared in Fig. 4.11. There we can see compared the electric potential and field distributions, the squared electric field versus the electric potential and

the particles densities. The first thing that has to be noticed by observing Fig. 4.11, is the astonishing agreement between the PIC simulation and the fluid model, meaning that our simulation is properly describing the physical behaviour of the system. Also, by observing the potential and field distributions in Fig. 4.11a and Fig. 4.11b, we can see that, once the ions are properly injected into the simulation, it does not appear any kind of source sheath. In those graphs it can also be seen that, the right hand side of the simulation is located at some point along the presheath, since the electric field there is negligible. This fact can be observed as well in Fig. 4.11d, where the quasineutral character of the presheath is recovered for $X \gtrsim 25$.

4.5. Conclusion

In this chapter we have described the implementation of the `CUPIC1D1V_PP` code, as well as the main results obtained with it. This code simulates the contact of an infinite planar probe with a plasma. Even though our main objective is to simulate the contact of a cylindrical probe with a plasma, this simulation will be the base code from which the cylindrical simulation will be developed. Also, the planar simulation has allowed us: to find the best initial conditions in order to quickly reach the steady state, and to develop a particle injection method that will have a great importance in the cylindrical simulation. Finally, the results provided by the simulation have been validated by comparing them with those obtained with a fluid model.

The following chapter will be devoted to explain the implementation of the cylindrical simulation, as well as the results obtained with it, which constitutes the main objective of this work.

Chapter 5

PIC simulation of a cylindrical Langmuir probe (CUPIC1D2V_CP)

5.1. Introduction

There are several reasons to study the behaviour of cylindrical Langmuir probes which are negatively biased with respect to the plasma. All these reasons have already been discussed in previous chapters, however they can be resumed into two. On the one hand, from an experimental point of view, the cylindrical geometry is widely used for its convenience and ease of manufacturing. On the other hand, the interest in using the ion saturation zone of the $I - V$ characteristic curve is motivated because of the negligible perturbation of the plasma that is produced.

Nevertheless, as we saw in Chapter 2, there are several models that predict the ion current collected by the probe, all of which can be classified into: orbital or radial theories. The problem being that each one produces different results when used to diagnose a plasma. So, there exists an interest in the plasma physics community to know which model better describes the ion current collected by the probe depending on the experimental conditions.

The complete kinetic description that PIC simulations provide, will allow us to shed light into the aforementioned problem, which has been the main objective and motivation of the present work. In this sense, the present chapter is devoted to explain the main parts of the developed simulation of a cylindrical Langmuir probe, `CUPIC1D2V_CP`, as well as the results concerning the transition from radial to orbital models that has been observed with it.

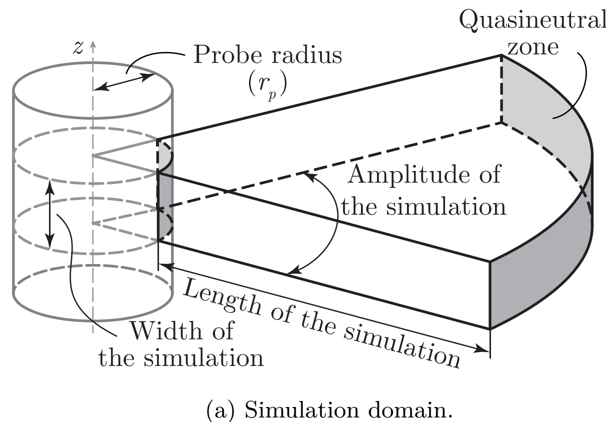
We will showcase the main parts of the code by explaining the differences with the code `CUPIC1D1V_PP`, which was explained in the previous chapter and in which the code `CUPIC1D2V_CP` is based on. Then, the results obtained with the simulation will be shown by means of Sonin-plots, in order to compare them with the results provided by the OML and ABR theoretical models.

As we did with the `CUPIC1D1V_PP` code, we are not going to provide here an exhaustive description of the simulation. However, the interested reader can check out the complete source code of the simulation, which can be found in Appendix C. The meaning of the name given to the software is analogous to the one in the previous chapter: “CUPIC” stands for CUDA PIC, “1D2V” indicates the dimensionality of the simulation and, “CP” stands for Cylindrical Probe.

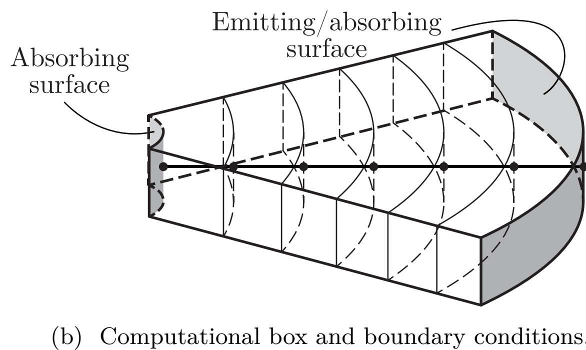
5.2. Computational abstraction of the system

Just like in the simulation of the planar probe, the first thing we have to define, in order to develop a simulation, is the physical space that we are going to simulate, *i. e.* the simulation domain. The answer is very similar to the one we gave in the previous chapter, since we want to simulate the space between a cylindrical Langmuir probe and an unperturbed plasma. Also, to avoid end corrections, we are going to consider the probe to be infinitely long. This causes the space between the probe and the

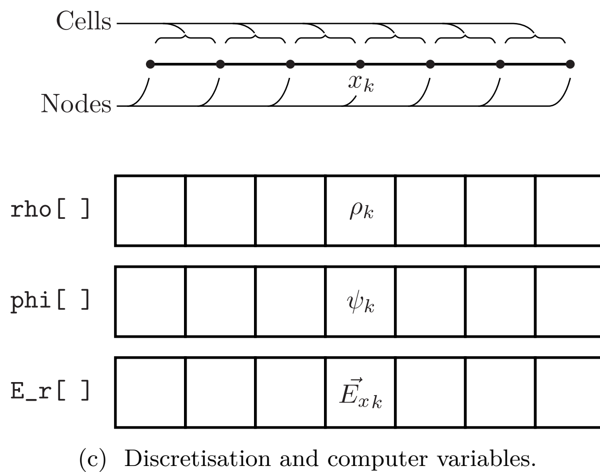
plasma to become infinitely large as well. However, when the probe is considered infinitely long, so the end corrections are neglected, the axial dimension becomes superfluous, and the motion of particles can be constricted to a plane perpendicular to the probe axis.



(a) Simulation domain.



(b) Computational box and boundary conditions.



(c) Discretisation and computer variables.

Figure 5.1: Computational abstraction of the simulation domain between a cylindrical Langmuir probe and a neutral plasma.

By taking into account the previous considerations, the simulation domain should be an annular disk, whose inner radius would be given by the probe radius and the outer one should be located in the quasineutral zone. However, as we already stablished and considered when solving fluid models in Chapter 2, the problem we are dealing with has cylindrical symmetry. That is, all the physical magnitudes involved in the problem only depends on the radial direction, *i. e.* the distance to the probe axis. For this reason, instead of considering the complete annulus around the probe, our simulation domain is defined as an annular sector.

The previously described simulation domain can be seen in Fig. 5.1a. There we can see that instead of the whole annulus, a small slab is considered. This allows us to restrict the amount of particles in the simulation domain to a reasonable number. Actually, there are two parameters that are adjusted so that the number of particles in the simulation domain does not become overwhelming: the width and the amplitude of the simulation. These two parameters control the total volume and thus the number of particles that are simulated. For example, we could consider the amplitude of the simulation to be 360°, however, in order to have a moderate number of particles the width of the simulation should be really small.

As we have already said, the only relevant dimension in our simulation is the radial one. For this reason, the computational box is a segment, as in the planar case. This computational box can be seen in Fig. 5.1b. It has to be noticed that, even though the computational box is a segment, in Fig. 5.1b the simulation domain is superimposed for the sake of ease of view. There are also represented the boundary conditions of the computational box, which are the same than in the planar case, except for the fact that the surfaces where they are implemented are different.

Also, just like in the previous chapter, the computational box is gridded in order to solve Poisson's equation. So finally, the computational box is discretised and represented by a bunch of nodes and the corresponding cells, as can be seen in Fig. 5.1c. In the computer, this abstraction is represented by the three arrays which are schematised in Fig. 5.1c. These arrays store the values of the three relevant macroscopic quantities at node positions, *i. e.* charge density, electric potential, an radial component of the electric field.

As can be observed by comparing Fig. 5.1 and Fig. 4.1, even though the geometries are very different, the computational box and the computer variables representing it, are the same in the planar

and cylindrical probe cases. However, there is a significant difference between both cases. In Fig. 4.1b we can see that, for the case of planar geometry, the volumes corresponding to the cells are all the same across the whole computational box. However, in Fig. 5.1b we can see that, for the case of cylindrical geometry, the volumes of each cell are different. Specifically, the volumes of the cells decrease as the left hand side of the simulation, *i. e.* the probe surface, is approached. This fact was already shown in Fig. 2.7 in Section 2.4.1, where we stated that the decrease of volumes as the probe surface is approached represents a presheath mechanism due to the geometry instead of ionisation. This fact has to be considered in the particle weighting step of the simulation, as well as during the iparticle injection, as we will see.

Once the simulation domain and the computational box are determined and characterised, it is time to describe the system of particles in it. The first thing that we have to notice is that the dimensionality of the cylindrical probe simulation is no longer 1d1v, instead we must consider two components of the velocity, *i. e.* 1d2v. For the spatial dimension we are considering the distance from the probe axis, while for the velocity we are considering the radial as well as the azimuthal components. The axial component of the velocity is superfluous since the probe is considered to be infinitely long. However, the azimuthal component must be taken into account, since, in order to consider the motion of particles restricted to the radial dimension, a centrifugal force term, which depends on the azimuthal component of the velocity, has to be considered. As we will see, this has to be carefully considered when developing the particle mover step.

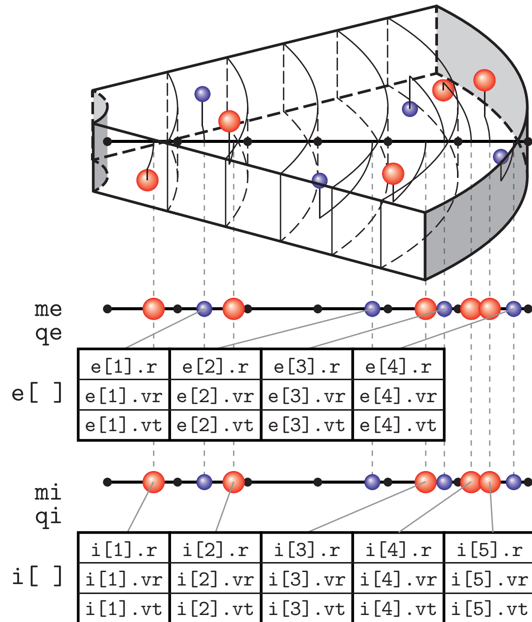


Figure 5.2: Computational abstraction of the particle system present in the simulation domain.

In Fig. 5.2 the computational abstraction of the particle system is schematised. There, we can see that the number of attributes considered for each particle is 5. On the one hand, we have 4 attributes which are just the same than the ones in the planar probe simulation: the mass of the particle, its charge, its position and its component of the velocity towards or backwards the probe surface. On the other hand, now we have another extra attribute for the azimuthal component of the velocity. Just like in the previous chapter, we are considering two ensembles of particles, one for electrons and another for ions. Each ensemble is characterised by the corresponding values of mass and charge, and a `particle` array that stores the position and both components of the velocity for each particle. Obviously, the definition of the `particle` struct, which can be found in Code C.14 (lines 29-34), has to consider the extra component of the velocity.

Finally, once that we have defined the simulation domain and the particle system that we are going to simulate, as well as their computational abstractions, we have that the main variables in our simulation are:

- To characterise the mesh: `rho[]`, `phi[]` and `E_r[]`.
- To characterise the particles: `me`, `qe`, `e[]`, `mi`, `qi` and `i[]`

The previous variables are almost equal to the variables in the previous chapter, being the only difference the new definition of the `particle` structure. Also, as we mentioned in the previous chapter, the names of these and all the variables that appear in the source code in Appendix C, are usually preceded by `d_`, `h_`, `g_`, `sh_` or `reg_`, meaning that the variable is allocated in the device, host, global, shared or register memory spaces, respectively.

Once we have described the computational abstraction of the system, we can showcase the main traits of the principal parts of the simulation. However, code of CUPIC1D2V_CP is based in the code of CUPIC1D1V_PP, and only small details are different between both of them. For this reason, the following section will be devoted to highlight the differences between both codes.

5.3. Differences between CUPIC1D2V_CP and CUPIC1D1V_PP

As we have already mentioned, most of the code of the simulation of the cylindrical probe is shared with the simulation of the planar probe. In particular, all the kernels defined in CUPIC1D2V_CP have the same execution configuration that their counterparts in CUPIC1D1V_PP. Actually, these kernels execute almost the same code except for a few differences that we are going to highlight in this section. However, the complete sources of the simulation found in Appendix C. There, it is also provided a makefile that automates the compilation process, shown in Code C.17, as well as an example of the input file from which the simulation reads its parameters, shown in Code C.18.

5.3.1. Initial conditions

There is not much to say about the initial conditions used in CUPIC1D2V_CP. The same behaviour with respect to the initial conditions is observed in the cylindrical probe simulation than in the planar one. When the system is initialised with particles in it, those particles tend to end up in trapped trajectories in potential wells, whose population decrease very slowly in time, and as a consequence, the transient state lasts longer. As our study is focused in the steady state, particularly we pursuit the value of the steady current collected by the probe as we will see, we are interested again in the initial conditions that reach the steady state the fastest. As we saw in Section 4.3.1, the fastest way to reach the steady state is to initialise the simulation with an empty system. In this way, as the simulation is filled with particles, they acquire the configuration corresponding to the steady state.

So, for the sake of performance, the initial conditions considered for the simulation have been those of an empty system. This can be seen in Code C.2 (line 141), where the initial number of particles is set to zero, while in the CUPIC1D1V_PP code two options where programmed, *i. e.* empty and filled system, as can be seen in Code B.2 (lines 140-142).

5.3.2. Particle weighting

In the particle weighting step there is a small modification with respect to the code of the planar probe simulation. The shape function considered is again the one corresponding to the CIC/PIC scheme, *i. e.* the function defined in Eq. (3.17), where the coordinates x and x_l are considered to be the distance from the probe axis to the particles and the nodes respectively. However the main difference is due to the cell volumes. Let us remember that, when the particle weighting step was explained in Section 3.3.1, for the sake of simplicity, we assumed a regular mesh and thus a constant cell volume across the whole grid, being V_c this volume as can be seen in Eq. (3.7) and Eq. (3.8). However, since the volume associated with each node/cell in our simulation is no longer constant (see Fig. 5.1b), the previously referenced equations have to be slightly modified.

By taking into account the previous considerations, the contribution of the l-th particle to the charge density of the $\vec{k}$-th node, can be expressed in the most general way as:

$$\rho^l_{\vec{k}} = \frac{q_l}{V_{\vec{k}}} S\left(\vec{r}_{\vec{k}} - \vec{r}_l\right) \tag{5.1}$$

q_l being the charge and $S\left(\vec{r}_{\vec{k}} - \vec{r}_l\right)$ the shape function associated with the l-th particle. Also, $V_{\vec{k}}$ is the volume associated with the $\vec{k}$-th node. So the complete charge density associated with the $\vec{k}$-th node is:

$$\rho_{\vec{k}} = \sum_{l=0}^{N} \frac{q_l}{V_{\vec{k}}} S\left(\vec{r}_{\vec{k}} - \vec{r}_l\right) \tag{5.2}$$

So, the main difference between the codes of CUPIC1D1V_PP and CUPIC1D2V_CP is that, in the former, the volume associated with each node was constant (Code B.4 lines 186-195), while, in the latter, the volume associated with each node is evaluated independently (Code C.4 lines 194-203)

5.3.3. Poisson solver

The differences in the Poisson solver part of the code in the cylindrical probe simulation with respect to the planar probe simulation, come from the fact that the expression of the Laplace operator, which appears in Poisson's equation, is different when expressed in cylindrical or cartesian coordinate systems.

It has to be noticed that, when explaining the force evaluation method in PIC simulations in Section 3.3.1, for the sake of simplicity, again, cartesian coordinates where assumed. By doing so, the resolution of Poisson's equation can be turned into the resolution of the system of linear equations given by Eqs. (3.20a), which leads the Jacobi method shown in Eqs. (4.1). However, when a cylindrical coordinate system is considered (r, θ, z), Poisson's equation is written as follows:

$$\begin{aligned}\nabla^2\varphi(\vec{r}) &= \frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial\varphi(\vec{r})}{\partial r}\right) + \frac{1}{r^2}\frac{\partial^2\varphi(\vec{r})}{\partial\theta^2} + \frac{\partial^2\varphi(\vec{r})}{\partial z^2} \\ &= \frac{\partial^2\varphi(\vec{r})}{\partial r^2} + \frac{1}{r}\frac{\partial\varphi(\vec{r})}{\partial r} + \frac{1}{r^2}\frac{\partial^2\varphi(\vec{r})}{\partial\theta^2} + \frac{\partial^2\varphi(\vec{r})}{\partial z^2} = -\frac{\rho(\vec{r})}{\varepsilon_0}\end{aligned} \tag{5.3}$$

By considering the symmetry reasons already argued multiple times, the derivatives with respect to θ and z are null. So, Poisson's equation is finally written as:

$$\frac{d^2\varphi(\vec{r})}{dr^2} + \frac{1}{r}\frac{d\varphi(\vec{r})}{dr} = -\frac{\rho(\vec{r})}{\varepsilon_0} \tag{5.4}$$

The derivatives appearing in Eq. (5.4) can be substituted by the following second order centred finite difference approximations:

$$\left.\frac{d\varphi(\vec{r})}{dr}\right|_{r_k} \simeq \frac{\varphi_{k+1} - \varphi_{k-1}}{2h} \tag{5.5a}$$

$$\left.\frac{d^2\varphi(\vec{r})}{dr^2}\right|_{r_k} \simeq \frac{\varphi_{k+1} - 2\varphi_k + \varphi_{k-1}}{h^2} \tag{5.5b}$$

h being the spacing of the mesh, and k the subindex which indicates the value at the k-th node.

Now, by taking into account the approximations in Eqs. (5.5), we can transform the differential equation in Eq. (5.3) into the following system of linear equations:

$$\varphi_k = \frac{1}{2}\left[\frac{h^2\rho_k}{\varepsilon_0} + \varphi_{k+1}\left(1 + \frac{h}{2r_k}\right) + \varphi_{k-1}\left(1 - \frac{h}{2r_k}\right)\right] \quad ; \quad \forall k = 1, \ldots, N_n - 2 \tag{5.6}$$

N_n being the number of nodes in the mesh.

Finally, the system of equations given by Eqs. (5.6) can be solved with the Jacobi method. In order to do so, successively improving approximations of the potential at node positions are obtained by using the following expression:

$$\varphi_k^{\text{new}} = \varphi_k^{\text{old}} + \frac{1}{2}\left[\frac{h^2\rho_k}{\varepsilon_0} + \varphi_{k+1}^{\text{old}}\left(1 + \frac{h}{2r_k}\right) + \varphi_{k-1}^{\text{old}}\left(1 - \frac{h}{2r_k}\right)\right] \quad ; \quad \forall k = 1, \ldots, N_n - 2 \tag{5.7}$$

This can be seen in the definition of the `jacobi_iteration()` kernel in Code C.4 (lines 277-280), where Eqs. (5.7) are used. Instead, in the code of `CUPIC1D1V_PP`, Eqs. (4.1) were used, as can be seen in Code B.4 (lines 243-245). Also, let us to notice that, in Eqs. (5.6) and Eqs. (5.7), the first and last nodes of the mesh are not considered since their values are fixed by the boundary conditions, as we already stated in Section 4.3.3.

5.3.4. Particle mover

In the previous sections, we have seen some minor changes in the code of `CUPIC1D2V_CP` with respect to the code of `CUPIC1D1V_PP`, which are strictly due to the fact that we have changed the geometry of the simulation from cartesian to cylindrical. Nevertheless, the changes in the particle mover, however minor as well, are due to more physical than geometrical reasons.

Let us start by remembering that, when developing these simulations, the planar and the cylindrical one, in order to avoid end corrections, we have considered the probes to be infinitely large. In the planar geometry, this means that there are two dimensions which become infinite and thus superfluous, so the motion of particles can be described with only one dimension. However, in the case of cylindrical geometry, in order to neglect the end corrections, we only have to consider one dimension to be infinitely large, *i. e.* its length. This means that, the axial dimension becomes superfluous and the motion of particles can be described in the plane perpendicular to the probe, as we mentioned in Section 5.2. Nevertheless, as we have mentioned previously, we are only considering one component of the position of the particles, *i. e.* their distance to the probe axis. In order to do so, and still consider the orbital motion of particles around the probe, a centrifugal force term must be taken into account. This centrifugal force depends on the azimuthal velocity and takes account of the fact that, for particles with high azimuthal velocity, it is harder to approach the probe and vice versa. This is also the reason why two components of the velocity are considered in the simulation, while only one component of the position is taken into account.

Now, bearing in mind the previous considerations, let us see how the equations of motion should be integrated. First, the equation of motion that has to be integrated is, as always, the second Newton's law, in the form shown in Eq. (3.3). However, contrary to the case of the planar probe, the force is not only due to the electrostatic force, since the centrifugal force must also be taken into account. So, Eq. (3.3) can be written as:

$$q_l \vec{E}_l + \vec{F}_{cl} = m_l \frac{d^2 \vec{r}_l}{dt^2} \; ; \quad \forall l \in [1, N] \tag{5.8}$$

$\vec{F}_{cl}$ being the centrifugal force suffered by the l-th particle, which can be written in terms of its azimuthal velocity and its distance to the probe axis as:

$$\vec{F}_{cl} = m_l \frac{v_{\theta l}^2}{r_l} \vec{u}_r \tag{5.9}$$

As we see in Eq. (5.9), the centrifugal force term only has a radial component. And, because of the symmetry of the problem, the electric field only has a radial component as well. So, finally, the equation of motion in the radial dimension, for the l-th particle, can be written as:

$$q_l E_l + m_l \frac{v_{\theta l}^2}{r_l} = m_l \frac{d^2 r_l}{dt^2} \tag{5.10}$$

which, by introducing the radial component of the velocity for the l-th particle, v_{rl}, can be split into two first order differential equations as follows:

$$\frac{dr_l}{dt} = v_{rl} \tag{5.11a}$$

$$\frac{dv_{rl}}{dt} = \frac{1}{m_l} \left(q_l E_l + m_l \frac{v_{\theta l}^2}{r_l} \right) \tag{5.11b}$$

As we can see in Eq. (5.11b), the integration of the radial component of the velocity, v_{rl} depends on the azimuthal component of the velocity, $v_{\theta l}$. Eqs. (5.11) could be easily integrated by using the same leap-frog scheme that we used in the simulation of the planar probe. However, in order to do this, we need and equation to describe the evolution of $v_{\theta l}$. It has to be noticed that, even though there are no forces in the azimuthal direction, the azimuthal component of the velocity is not a constant of motion. The reason is that, since we are describing the motion of a particle in a plane, the angular momentum of the particle must be conserved, and so:

$$J_l = m_l r_l v_{\theta l} \equiv \text{const.} \Rightarrow r_l(t_0) v_{\theta l}(t_0) = r_l(t) v_{\theta l}(t) \tag{5.12}$$

Once that we have equations to describe the evolution of the position and the two components of the velocity that are being considered, it is easy to implement the leap-frog integration scheme, only one more thing has to be considered. Since the azimuthal component of the velocity appears in the force term of the second Newton's law, *i. e.* right hand side of Eq. (5.11b), instead being evaluated at half time steps, like the radial component, it has to be evaluated at full time steps, like the electric field or

the position, see Fig. 3.8. So the integration of the equations of motions would be like:

$$ {v_{rl}}^{p+1/2} = {v_{rl}}^{p-1/2} + \frac{\Delta t}{m_l}\left(q_l {E_l}^p + m_l \frac{{v_{\theta l}^2}^p}{{r_l}^p}\right) \tag{5.13a} $$

$$ {r_l}^{p+1} = {r_l}^p + \Delta t\ {v_{rl}}^{p+1/2} \tag{5.13b} $$

$$ {v_{\theta l}}^{p+1} = \frac{{r_l}^p {v_{\theta l}}^p}{{r_l}^{p+1}} \tag{5.13c} $$

where Eq. (5.13c) is obviously derived from Eq. (5.12).

The actual implementation of Eqs. (5.13) into the particle mover module of the simulation can be seen in the definition of the `leap_frog_step()` kernel, which appears in Code C.6 (lines 54-107).

5.3.5. Particle injection

In the particle injection code of the CUPIC1D2V_CP simulation, there are a two main differences with respect to the corresponding code of the CUPIC1D1V_PP simulation. One of them is due to the different dimensionality of the simulation, 1d2v versus 1d1v, while the other is due to the fact that a different presheath mechanism is present in the cylindrical probe case.

Let us start by remembering that Eq. (3.34), which gives us the flux of particles crossing a certain surface and thus allows us to evaluate the number of particles entering into the simulation each time step, is valid whatever it is the geometry. Even though in Section 3.3.3 we considered a cartesian coordinate system, the only restriction to the calculations performed there was that, the drift velocity of particles must be perpendicular to the surface through which the flux is going to be evaluated. As we have previously said multiple times, in the case of cylindrical probes, due to its symmetry, the dependence of all macroscopic quantities is restricted to the radial dimension. That is the case of the electric potential, reason why the potential drop and thus the acceleration originated in the quasineutral zone, is produced in the radial dimension. So, in the quasineutral zone, particles will be described by the following distribution function:

$$ f(\vec{v}) = n\left(\sqrt{\frac{m}{2\pi k_B T}}\right)^3 \exp\left(-\frac{m}{2k_B T}((v_r - v_d)^2) + v_\theta^2 + v_z^2\right) \tag{5.14} $$

Now, as can be seen in the right hand side of the computational box in Fig. 5.1b, the surface through which particles enter into the simulation is perpendicular to the radial direction. So Eq. (3.34) can be used to evaluate the flux of incoming particles. Once we know this, let us focus in the differences with the previous code.

On the one hand, in CUPIC1D2V_CP we are considering one extra component of the velocity with respect to CUPIC1D1V_PP, which consequently has to be initialised for each particle entering the simulation. The different components of the velocity of incoming particles are initialised by obtaining random numbers from the corresponding probability distribution functions. We know that, the probability distribution function of the incoming particles, is given by $f_{\text{inc.}}(\vec{r}) \propto v_r\ f(\vec{r})$. This distribution can be factorised into three different distributions, one for each component of the velocity. So, the probability distribution for the radial component of the velocity is given by the following Rayleigh distribution:

$$ f_{\text{inc.}}(v_r) = v_r \sqrt{\frac{m}{2\pi k_B T}} \exp\left(-\frac{m(v_r - v_d)^2}{2k_B T}\right) \tag{5.15} $$

which is the same distribution as the one corresponding to the component of the velocity perpendicular to the planar probe, *i. e.* the one we were considering in CUPIC1D1V_PP. Then, the other components of the velocity, being the azimuthal the only one relevant for us, will be described by Gaussian distribution functions. Specifically, the azimuthal component will be described by:

$$ f_{\text{inc.}}(v_\theta) = \sqrt{\frac{m}{2\pi k_B T}} \exp\left(-\frac{m v_\theta^2}{2k_B T}\right) \tag{5.16} $$

On the other hand, the calibration method for the drift velocity of ions that we explained in Section 4.3.5, is slightly different. There, we imposed an arbitrarily small value for the electric field at the sheath

edge. This was done in order to find a value for the drift velocity that ensures that quasineutrality is achieved at the right hand side of the simulation. That method worked incredibly well, as it was confirmed by the results shown in Fig. 4.11. However, it would be desirable to have a value for the electric field at the sheath edge that it is somehow related to the physics of the system that is being simulated. The only way to introduce such information is by considering a value of the electric field provided by the quasineutral solution of a fluid model.

As we stablished in Section 4.3.5, the presheath, or quasineutral zone, rely in the existence of a presheath mechanism, by means of which the ion flux can increase. For this reason, the aforementioned approach was not implemented in the planar probe simulation, since the usual presheath mechanism in planar geometry is the ionisation, which is not taken into account by our simulation. This can be seen in the quasineutral solution given by Eq. (1.26), which depends on the ionisation rate. However, in the cylindrical probe case, the presheath mechanism is the cylindrical geometry itself, as it was noticed in Section 2.4.1. For this reason, the quasineutral solution does not depends on parameters that are not taken into account by our simulation. This can be seen in the quasineutral solution provided by Eq. (2.68) and Eq. (2.69), which can be combined in order to obtain the following expression for the electric field in the quasineutral zone:

$$E(R) = \frac{2}{R}\frac{(\psi(R) - 2\beta)e^{2\psi(R)} + 2\beta e^{3\psi(R)}}{(1 + 2\psi(R) - 4\beta)e^{2\psi(R)} + 6\beta e^{3\psi(R)}} \tag{5.17}$$

The implementation of the quasineutral solution given by Eq. (5.17) can be seen in the code of the `calibrate_ion_flux()` function in Code C.8 (lines 193-249).

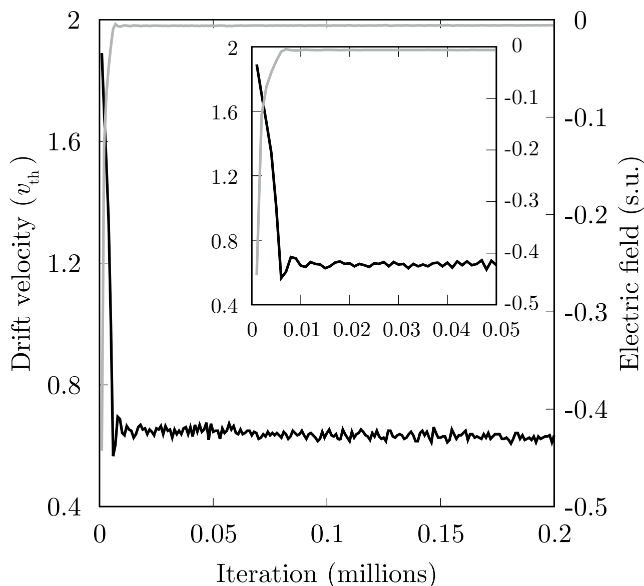


Figure 5.3: Calibration of the ion flux by considering the quasineutral solution in Eq. (5.17). Black solid line represents drift velocity and grey solid line represents the electric field at the sheath edge.

In Fig. 5.3 the calibration process for the ion drift velocity is shown. Just as in the case of the planar probe simulation, it can be seen that as the drift velocity is adjusted the electric field at the sheath edge becomes negligible. The main difference with respect to CUPIC1D1V_PP is that, the steady value reached by the electric field at the sheath edge, is in accordance with the quasineutral solution given by Eq. (5.17).

It also has to be noticed that, in Fig. 5.3, instead of using the Bohm velocity as the unit for the drift velocity, like it was done in Fig. 4.10, we have chosen to use the thermal velocity of ions. The reason being that, by doing so, we can easily see the drift velocity to thermal velocity ratio reached at the steady state of the simulation. This quantity is interesting since, by taking into account the graph in Fig. 3.10, it allows us to understand if the ion influx during the steady state is dominated by the drift or thermal velocities.

However, it should be highlighted that, despite the fact of using different units in Fig. 5.3 and Fig. 3.10, the drift velocity is on the same order of magnitude in both cases. Actually, there is not much difference between both scales, as can be seen from the values of the Bohm and thermal velocities in the dimensionless units we have been using:

$$V_B = \sqrt{\frac{1}{\gamma}} \tag{5.18a}$$

$$V_{\text{th}} = \sqrt{\frac{2\beta}{\gamma}} \quad \text{(for ions)} \tag{5.18b}$$

For example, in the simulation whose evolution is shown in Fig. 5.3 the ion to electron temperature ratio was $\beta = 0.1$. These values yield a ratio $V_{\mathrm{th}}/V_B \simeq 0.45$.

5.4. Hybrid code optimisation

In this section we are going to explain a significant optimisation introduced in the CUPIC1D2V_CP code. This optimisation was introduced mainly in order to speedup the simulation, reducing the computational times and thus accelerating the obtention of results. By means of the aforementioned optimisation, we have been able to perform a huge number of simulations, which would have been otherwise impossible due to time restrictions. It has to be noticed that, in the code shown in Appendix C the optimisation is already implemented. For this reason, if the code of CUPIC1D2V_CP has been thoroughly checked out, the reader has probably noticed what we are going to discuss in here.

Let us start by stating a couple facts:

- On the one hand, even though our interest has been to simulate and study the behaviour of Langmuir probes in the ion saturation zone of the $I - V$ characteristic curve, *i. e.* $\psi_p \ll 0$, the simulations that we have developed are not restricted to those conditions. In particular, the biasing potential of the probe can be set to any value in the input file of the simulations, as can be seen in Code B.18 (line 14) and Code C.18 (line 17).
- On the other hand, the main reason of the huge computational time required by PIC simulations is that, in order to not to accumulate huge numerical errors during the integration of the equations of motion, the time step of the simulation should be small enough, so that the spatial motion of the fastest particles during one time step is small when compared to the mesh spacing. Due to its higher temperature and smaller mass, electrons have much more mobility than ions, and thus are the fastest particles in our simulations. Actually, this is one of the reasons why the time units of the simulation are defined as the inverse of the electron plasma frequency. Let us remember that this electron plasma frequency was introduced at the end of Section 1.3 and, its definition can be seen in Eq. (1.9c). So, our simulations must advance with a time step suited for the fast electrons, which results in a time step ridiculously small for the motion of ions. As a consequence, a large number of time steps has to be performed so that ions move a little bit. Obviously, the simulation does not reach its steady state until the slowest specie, *i. e.* ions, reach it. That is the reason why in Section 4.3.1 the phase space of ions was used in Fig. 4.3 to determine when the steady state was reached.

Now we can answer the question: how do we reduce the computational time needed by our simulations in order to reach the steady state? Because of the second fact mentioned before, the previous answer can be changed by the following: how can we increase the time step of our simulations without compromising the precision with which electrons are described? Additionally, in order to answer this question, the first fact has to be taken into account. Since we are going to perform simulations where the probe is going to be high negatively biased, we can introduce modifications in our code that are only valid for such conditions. The actual modification that we introduced consist of describing the electrons in the simulation by using a fluid description, instead of describing them as particles, as explained in Section 5.2. By doing so, our simulation can no longer be considered a "pure" particle simulation, since it better fits under the hybrid simulation category shown in Fig. 3.1.

Let us find out if a fluid description for electrons can be introduced without loosing relevant information in our simulations. First, when in the ion saturation zone, from a fluid point of view, the complete description of electrons is readily known. As we have already stated multiple times during Chapters 1 and 2, when the biasing potential of the probe is negative with respect to the plasma, electrons are in thermal equilibrium with the electric field, so their distribution function is given by Eq. (1.2) or its cylindrical counterpart. Furthermore, as long as the biasing potential of the probe is negative enough, $\psi \leq -10$, their density distribution is also known and given by Eq. (1.3) in the planar case, or Eq. (2.44) in the cylindrical case. So, in the dimensionless units of the simulation, the electron density should be described by the following expression:

$$N_e(R) = e^{\psi(R)} \tag{5.19}$$

So, instead of considering an ensemble of particles to describe electrons, as during the PIC simulation the electric potential is evaluated every single time step at node positions, Eq. (5.19) can be used to evaluate the electron density at node positions, and thus the charge density there due to electrons. Then, the charge density due to ions would be evaluated by the particle weighting algorithm previously explained, so Poisson's equation can be solved normally. Finally, with the values of the electric potential at the new time step, the electron density at the new time step can be evaluated, again with Eq. (5.19), without being needed the integration of the equations of motion for electrons. By using the previous scheme we can avoid the integration of the equations of motion for electrons, so a larger time step can be used, which will be best suited for the motion of ions. Obviously, by considering a larger time step, more physical time can be simulated with fewer time steps, so the obtention of results is effectively accelerated.

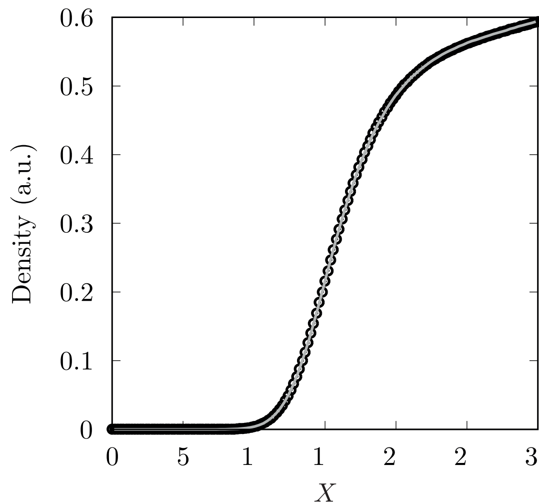


Figure 5.4: Comparison between the electron density by using particles (grey solid line) or the fluid approximation given by Eq. (5.19) (circles).

It can be probed, that the exact same results are obtained for the macroscopic quantities by simulating the electrons as particles or by using Eq. (5.19). For example, in Fig. 4.11d the particle densities corresponding to the PIC simulation where obtained by means of histograms of the particle positions, and we can compare this electron density with the one provided by Eq. (5.19) using the values of the electric potential obtained with the PIC simulation that are shown in Fig. 4.11a.

In Fig. 5.4 the previous comparison is shown. As can be seen, both methods provide the exact same electron density, and thus the same charge density, electric potential and forces suffered by the ions. So, electrons can be safely removed from the simulation, as long as their presence is considered by means of Eq. (5.19), and ions are not going to notice their absence. It has to be noticed that, for the sake of brevity, only the case in Fig. 5.4 is shown, however the same exact results are obtained by using the cylindrical simulation, other simulation conditions or even data obtained from the transient state.

Once we are sure that the hybrid optimisation works properly, the evaluation of the electron density was carried out by the function `virtual_to_grid()`, whose definition can be seen in Code C.4 (lines 215-238). There, the actual implementation of Eq. (5.19) can be seen at line 231. With this hybrid code, the number of iterations needed in order to reach the steady state is one ore two orders of magnitude smaller than with electrons as particles. This can be seen by comparing Fig. 5.3 with Fig. 4.10. In the former, with the hybrid optimisation implemented, we can see that the drift velocity reaches its steady state after 50000 iterations approximately, while in the former, without this optimisation, more than a million iterations are needed.

5.5. Radial to Orbital motion transition

In this section we are going to show the results concerning the transition from a radial to an orbital behaviour of ions, in the surroundings of a negatively biased cylindrical Langmuir probe, that have been obtained with the `CUPIC1D2V_CP` simulation. These results, as it was stated in the objectives section of the document, constitute the main aim that motivated this work.

First, we are going to explain how the results provided by the simulation are going to be analysed. As we stated in Section 2.5, the Sonin-plot represents a powerful tool that allows us to identify if the motion of ions can be classified as radial, orbital, or something in between. Let us remember that, the Sonin-plot, was determined by the representation of two coordinates, $(x_{\text{sonin}}, y_{\text{sonin}})$, the definition of which where provided by Eq. (2.85) and Eq. (2.84). However, by using the dimensionless variables, that we have been

working with during the whole document, the coordinates of the Soning-plot can also be defined as:

$$y_{\text{sonin}}(R_p, \psi_p, \beta) = \frac{I^* \sqrt{\gamma}}{\sqrt{2\pi} N_{e0}} \frac{1}{R_p} \tag{5.20a}$$

$$x_{\text{sonin}}(R_p, \psi_p, \beta) = \frac{I^* \sqrt{\gamma}}{\sqrt{2\pi} N_{e0}} R_p \tag{5.20b}$$

where $I^* = i\lambda_D / e\omega_{pe}$, which is the parameter that we are going to measure from the raw data provided by the simulation. Also, in Section 2.5, we obtained the Sonin-plot representation corresponding to the OML model, given by Eqs. (2.88), as well as the ABR model, obtained by numerical integration and using Eqs. (2.90). The actual representation of both theories can be seen in Fig. 2.14.

Obviously, the Sonin-plot depends on the value of the biasing potential of the probe. In our case, $\psi_p = -25$ is the biasing potential chosen for all the simulations performed for the obtention of the results presented here. Also the ion to electron mass ratio was chosen to be $\gamma = 7296.0$, which is the value corresponding to Helium ions. This value was chosen because it is the gas with which J. M. Díaz-Cabrera *et. al.* [47] experimentally observed the transition from ABR to OML behaviours.

With these parameters, several simulations where performed for different values of the ion to electron temperature ratio, $\beta \in [0, 1]$, and dimensionless probe radius, $R_p \in [0.5, 4]$. It has to be noticed that, the previous ranges where considered in order to be representative for most of the low temperature and low pressure plasmas, which is the kind of plasmas on which our research is focused. Values $\beta \geq 1$ are rarely found except, for example, in fusion plasmas. Also, $R_p \leq 0.5$ or $R_p \geq 4$ values are either very small or very large for a cylindrical probe and, in those cases, the behaviour is mainly explained as OML or ABR as we will see.

First, we performed a set of simulations where different R_p values where considered for a fixed β values. These results can be seen in Fig. 5.5 for β =0.0, 0.1, 0.2 and 0.4. There we can see how, independently of the probe radius, the ABR model is the theory that properly describes the behaviour of ions in the limiting case $\beta \to 0$, while for $\beta \neq 0$ the behaviour of ions depends on the probe radius. In order to analyse those results, two regimes have to be defined: small probe radii ($R_p \lesssim 2$) and large probe radii ($R_p \gtrsim 2$).

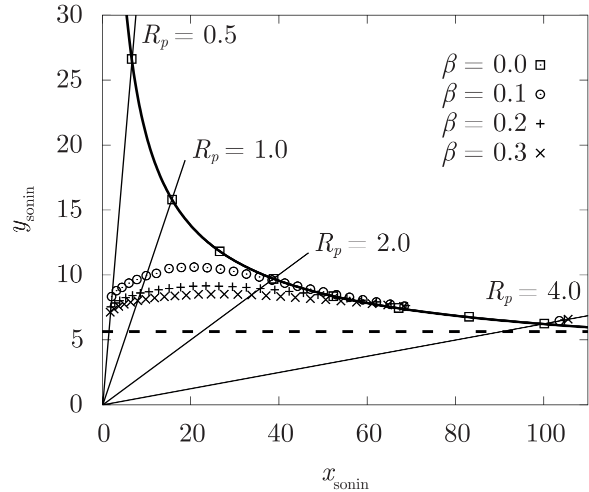


Figure 5.5: Dependence of the Sonin-plot on the dimensionless probe radius for different β values. ABR (thick solid), OML (thick dashed) and constant R_p (thin solid) are also shown.

On the one hand, for probe radii approximately smaller than 2, we can see that as we increase β, a transition is found from the ABR to the OML theory. This is due to a decrease in the dimensionless current collected by the probe, I^*, as the temperature of ions is increased with respect to the electron one. We can also see that, as the dimensionless probe radius is decreased, the transition becomes more pronounced.

On the other hand, for probe radii approximately larger than 2, we can see that as we increase β, the behaviour of ions can no longer be described by the ABR theory. The difference with the previous case is that the dimensionless current collected by the probe, I^*, does not decrease, as an orbital behaviour would suggest. Contrary to that, the current collected becomes larger than the prediction of the ABR theory. A current larger than the one obtained with the ABR model can be obtained with a radial theory that considers the thermal motion of ions, such as the model developed by Fernández Palop *et. al.* [14] that was reviewed in Section 2.4.2. So, we can conclude that for large probe radii the behaviour of ions remains mostly radial as β is increased.

So, as it can be seen in Fig. 5.5 and we have stated, once the dimensionless probe radius is fixed, the behaviour of ions is determined by the value of ion to electron temperature ratio. For this reason, a second set of simulations where performed later. In this case, the R_p value was fixed while β varies from 0 to 1.

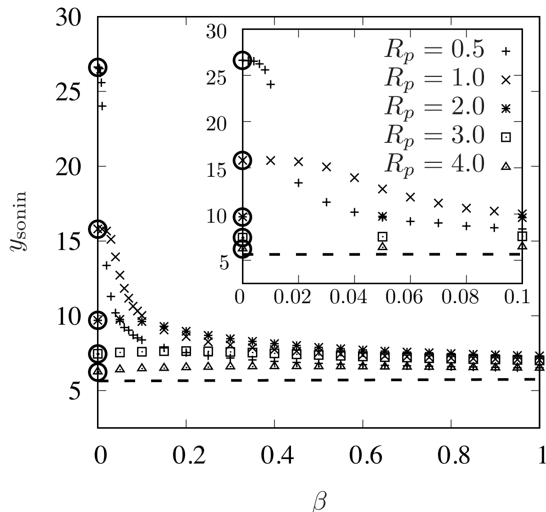


Figure 5.6: Dependence of the ordinate of the Sonin-plot on the ion to electron temperature ratio for different R_p values. ABR (circles) and OML (thick dashed) are also shown.

Since in this case we want to see the dependence on β, instead of using a regular Sonin-plot, we are going to show the results by plotting y_{sonin} versus β. In Fig. 5.6 the results of this set of simulations can be seen. There, the prediction of the OML model as well as the predictions of the ABR model for every probe radius are also shown. Obviously, since the ABR theory is only valid for the case $\beta = 0$, only those points are shown.

The first thing we notice from the results shown in Fig. 5.6 is what we have already seen in Fig. 5.5. That is, the radial theory is recovered as the ion to electron temperature ratio tends to zero, for every probe radii. This means that the radial model provides a better prediction when the ion temperature is negligible compared with the one of electrons, although collisions are not taken into account. On the other hand, for high ion temperature values the orbital theory provides a better description of the behaviour of ions in the surroundings of the probe. This is because as we increase β the angular momentum of ions also increases, so the fraction of ions that are able to orbit around the probe becomes dominant in the sheath dynamics.

We can conclude that between negligible and high ion temperatures, a transition from the radial behaviour to an orbital one is found. The transition is explained because the fraction of ions that orbit around the probe goes from negligible to dominant. Also, the dimensionless probe radius is found to be a key parameter on the way this transition occurs. This is reasonable, because the larger the probe radius the harder it is for an ion to orbit around the probe. Actually, as can be seen in Fig. 5.5 the aforementioned transition is only found for relatively small probe radii, *i. e.* $R_p \lesssim 2$. This behaviour can also be seen in the inset of Fig. 5.6, where it is clear that the transition from ABR to OML occurs for smaller β values as the probe radius becomes smaller. Also, as the probe radius increases and the probe becomes more planar-like, the transition gets less prominent, which is reasonable since the difference between both models also decreases.

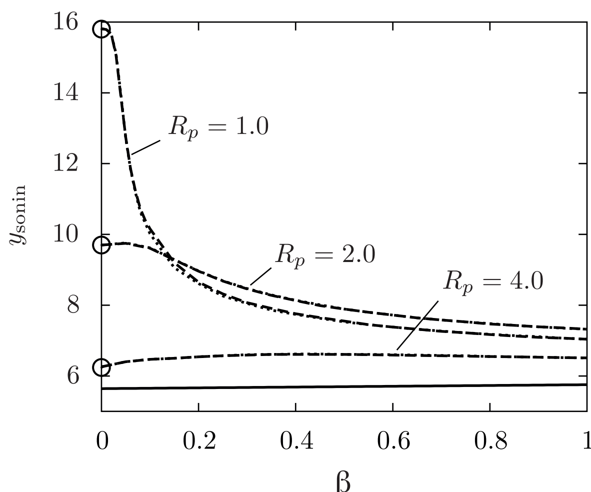


Figure 5.7: Dependence of the ordinate of the Sonin-plot on the ion to electron mass ratio for different R_p and β values. He^{+} (dotted lines), Ar^{+} (dashed lines), ABR (circles) and OML (solid line) are shown.

Finally, we decided to perform another set of simulations considering a different ion mass. The motivation for this last set of simulations was that, J. M. Díaz-Cabrera *et. al.* [54] recently informed that the transition that they found for the Helium plasma was not observed for the case of Argon. For this reason, we decided to perform also a set of simulations just like the previous one but for $\gamma = 72821.0$, which is the value corresponding to Argon ions.

In Fig. 5.7 we can see the results of the simulations for Helium plasmas compared to the ones for Argon. It is clear that the behaviour is exactly the same for both cases, which leads us to the conclusion that the transition between the radial and orbital behaviours is independent of the mass of the ions, and must be due to the different collisionality rates in Argon and Helium plasmas. Since our simulation properly describes the behaviour of a Helium plasma while not taking into account collisions, it means that in the case of Helium, probably due to its smaller size, collisions are negligible in the conditions studied by J. M.

Díaz-cabrera *et. al.*. However, the lack of transition in the Argon plasma means that, contrary to the Helium plasma, collisions can not be neglected. This is reasonable, since Argon has a smaller mean free path than Helium [55]

5.6. Conclusion

In this chapter we have described the implementation of the CUPIC1D2V_CP code by showing the differences with CUPIC1D1V_PP. This code simulates the contact of an infinite cylindrical probe with a plasma. We have also shown the results concerning the transition from radial to orbital motion of ions in the surroundings of a negatively biased cylindrical Langmuir probe, that have been obtained with it. This results have allowed the understanding of experimental results previously found by the research group where I have developed this work, and are in accordance to those reported by other authors like Chen [41].

With this chapter we finish the second and main part of the thesis, where we have described the simulations developed. We have also reported the results obtained with those simulations. The following part will be devoted to summarise this work as well as to state future outlooks, perspectives and ideas.

Part III

Final remarks & Conclusions

Chapter 6

Summary, contributions and future perspectives

6.1. Introduction

It is not easy to summarise all the work that has been carried out during a PhD research over more than four years. However, in this final chapter, I will try to put it in a nutshell. The structure of the chapter is going to be as follows:

- First, a brief chronological description of the activities that have been carried out is included. There, I comment the developments and activities performed during my research, as well as some of the problems found that have not been included in this document for the sake of clarity.
- Then, an enumeration of the main results obtained during this research is included. There, it is highlighted their relevance for the field of plasma physics in general, and the probe diagnosing techniques in particular.
- Finally, a compendium of future perspectives and ideas to continue the research started during my PhD stage is included. There I detail, among others, future expansions for the developed codes that we plan to accomplish.

Last, but not least, some final conclusions will be included to bring this work to a close.

6.2. Summary

I should start this section by stating something about the type of codes developed during this research, I mean, any kind of particle simulation. Even though they may look simple, at a first glance, because they are based in firsts principles simulations, they constitute complex pieces of software. Especially because of the, almost mandatory, requirement of parallelisation techniques. In general, parallel codes require the investment of extra time during their development stage. For this reason, parallel codes only worth the time needed for their development when: really big (computationally speaking) problems are being solved and the final program is going to be extensively used once developed. Both of these criteria are fulfilled in our case. Nevertheless, this does not mean that the extra time is not going to be needed, specially when developing in a relatively new platform, as CUDA.

Now, we can briefly describe the chronological progress of the work developed during my PhD:

- The first year was devoted to establish the state of the art about plasma diagnosing with Langmuir probes. In particular, the extensive bibliographic review about theories that predict the ion current collected by Langmuir probes shown in Chapter 2 was performed. I also familiarise myself with the previous research developed by the group where I have worked. Finally, even though I have

been learning different aspects of the CUDA framework and programming language till nowadays, mainly because it is a young platform still in development, during this first year was when I learned all the basics of CUDA and the GPGPU paradigm. Besides, I set up the workstation that I have been using all these years.

- During the second year, I started the development of the first simulation. It has to be noticed that, actually, this simulation has not been covered here. It was a 2d2v simulation of the contact of a planar probe with a plasma. In order to develop this code, I also learned all the basis about particle simulations, particularly what is related to PIC codes. This simulation was significantly more complex that `CUPIC1D1V_PP` or `CUPIC1D2V_CP`, the reason being that, because of the size of a two dimensional system, the use of sorting algorithms for the particles was mandatory. Sorting algorithms are not particularly complex by themselves, however they are not very well suited for their parallel execution. For this reason, the debugging of this code lasted several months, and even after that, the simulation crashed after several hundred thousands of iterations. Nevertheless, all the results provided by the simulation, before crashing, showed that the second dimension was superfluous and, no relevant information was lost by diminishing it.

- It was on the third year when the first versions of `CUPIC1D1V_PP` and `CUPIC1D2V_CP` were developed. First, the simulation of the planar probe was developed. This was done by modifying the code of the two dimensional simulation. The smaller system allowed us to remove the sorting algorithm from the code, and thus to avoid the problems that caused the simulation to crash. Then, the first test runs of the simulation where performed and several improvements were introduced. Among these, those explained in Chapter 4 related to the initial conditions and the appearance of a source sheath. Also, we found out that, under certain conditions the simulation became unstable. After considering several types of instabilities, physical and numerical, we realised that the problem was caused by streaming instabilities. This instabilities have a physical origin, they are due to the fact that a stream of particles, ions in our case, move with a drift velocity through a quasineutral zone. Several authors solve this problem by introducing artificially high collision rates in order to dampen the oscillations. However, we realised that the length of the simulation was the key factor and, instabilities do not appear if we do not try so simulate a longer than necessary quasineutral zone. Once the simulation was tested, by comparing their results with those provided by fluid models, the development of the first version of `CUPIC1D2V_CP` was carried out. The first preliminary results, related to the behaviour of ions in the surroundings of a cylindrical Langmuir probe, were also obtained at that time. Nonetheless, further improvements of the code were needed before obtaining the final results shown in Section 5.5.

- During the last year, several different things were performed:

  - On the one hand, I realised a three months stay with Prof. Tomaž Gyergyek at the University of Ljubljana (Slovenia). Specifically, the stay was performed in the Faculty of electrical engineering of the University of Ljubljana as well as the Reaktor Center of the Jožef Stephan Institute (IJS), also in Ljubljana, where Prof. Gyergyek develops his research. Thanks to this stay I was able to use the powerful cluster of GPUs available at the IJS Reaktor Center. Also, during the stay, Prof. Gyergyek's group and the group were I have developed my thesis have started a collaboration that hopefully will extend over the following years. This collaboration was forge on the base of the common interests shared by both groups. In particular, during my stay I developed a modification of the planar probe simulation in order to consider two populations of electrons with different temperatures. The development of this simulation was aimed towards the study of some effects predicted by a kinetic model previously developed by Prof. Gyergyek. Even though very promising results were obtained by the end of the stay, the simulation needs further improvements in order to provide results suited for their publication. Specifically, we think that a presheath mechanism has to be included in order to avoid the dependence of the results with the length of the simulation.

  - On the other hand, after finishing the stay and, enlightened by the results obtained there, the use of the presheath solution was introduced in `CUPIC1D2V_CP` code, as stated in Section 5.3.5. This modification allowed us to obtain, for the first time, conclusive results concerning the transition from radial to orbital behaviour of ions. However, in order to speedup the obtention of results, the hybrid code optimisation described in Section 5.4 was introduced. This was the last modification of the code and, after that, the results that have been shown in Section

5.5 were obtained. Those results were published as a letter in *Plasma Sources Science and Technology*, which is the second journal of its category (Physics, Fluids & Plasmas) at JCR.

Finally, the past months have been devoted to writing, revising and correcting the present document.

6.3. Contributions

The different results derived from the work presented here, have been mentioned along the text. However, in this section, we are going to enumerate them. Also, some brief comments about the relevance or these results are going to be included. So, without further ado, the following results have been obtained during the my PhD research:

- A 1d1v PIC simulation of the contact of an infinitely large planar Langmuir probe with a plasma has been developed, `CUPIC1D1V_PP`. In the simulation, the probe is assumed to be perfectly absorbing and, its biasing potential with respect to the plasma can be fixed or leaved floating. The simulated plasma consist of electrons and singly ionised ions and, collisions have not been taken into account. The code has been developed under the CUDA framework for its execution in GPGPU environments, allowing a great reduction of the execution times of the simulation with respect to a sequential code.
- A wide variety of initial conditions have been tested with `CUPIC1D1V_PP`, finding out that the initial conditions which fastest reach the steady state consist in an empty system. This is true, except in the case that the exact distribution function for the different particles at the steady state, is known beforehand, which is not usual at all. This statement might be applicable to any open particle simulation.
- An original method for the injection of ions has been implemented into the code of `CUPIC1D1V_PP`. The method is based in the fact that quasineutral conditions, *i. e.* negligible electric field, have to be fulfilled at the particle source. With this method, the appearance of a source sheath is avoided, also the Bohm velocity is obtained. With this method implemented, the results provided by `CUPIC1D1V_PP` are in great agreement with those provided by fluid models under simple conditions.
- A 1d2v PIC simulation of the contact of an infinitely long cylindrical Langmuir probe with a plasma has been developed, `CUPIC1D2V_CP`. In the simulation, the probe is assumed to be perfectly absorbing and its biasing potential with respect to the plasma can be fixed or leaved floating. The simulated plasma consist of electrons and singly ionised ions and collisions have not been taken into account. In this simulation, the same injection method developed for `CUPIC1D1V_PP` has been implemented. The code has been developed under the CUDA framework for its execution in GPGPU environments, allowing a great reduction of the execution times of the simulation with respect to a sequential code.
- A fluid approximation has been used to describe the electrons in `CUPIC1D2V_CP`. This approximation is only valid for the case of a high negatively biased probe. However, it has allowed us to further decrease execution times, to approximately a few hours, in order for a timely obtention of results.
- The aforementioned PIC simulation has been used to study the transition from a radial to an orbital behaviour of ions in the surroundings of a cylindrical Langmuir probe. The results obtained from the simulation describe the same transition previously found, by the group where I have developed my PhD, in an helium plasma. It has been found that the ion to electron temperature ratio is the parameter that explains this transition. It has also been found that the dimensionless probe radius greatly affects the shape of the transition, which only occurs for relatively small probe radii. These results will enlighten the pursuit of a theoretical model that includes both, the radial and orbital theory, as limiting cases. Such model would allow a precise diagnose of plasmas by using Langmuir probes in the ion saturation zone of the characteristic for any plasma conditions. For these reason, this result has been published [56] as a a letter in *Plasma Sources Science and Technology*, which is the second journal of its category (Physics, Fluids & Plasmas) at JCR.
- `CUPIC1D2V_CP` has also been used to study the dependence of the previous result on the ion mass. It has been found that this parameter does not affect at all to the transition from radial to orbital

behaviour. Since the lack of this transition has been observed by our group for Argon plasmas, this means that collisions, less frequent in Helium plasmas due to their larger collision mean free path, must be responsible for this behaviour.

Finally I would like to notice that, even though it may not look like that, the development of the codes `CUPIC1D1V_PP` and `CUPIC1D2V_CP` is the main result from this thesis. Not only because most of the time of this thesis has been devoted to its development, but also because, once developed, these simulations will allow us to study a lot of different problems by introducing small modifications into the code. The transition from radial to orbital behaviour of ions being the first problem tackled with them. More about the expandability of the simulations will be said in the following section.

6.4. Future perspectives

As we have said in the previous section, the codes developed during this thesis have been designed with "upgradability" in mind. For this reason, a long term goal of this work is: to develop a set of codes that allow the simulation of the behaviour of different Langmuir probes under diverse conditions and considering a variety of processes. That is why, this thesis represents only the first stage of a project where we will keep working in the foreseeable future. Some of the ideas that we would like to tackle next are:

- Without the need of introducing any modification in the codes that we have presented here, we can obtain several different results: floating potentials, $I - V$ characteristic curves, ion velocity distribution functions along the sheath, transient states of a cavity filled with plasma, transient states after a sudden change in the biasing potential,...
- The introduction of collisional processes in the `CUPIC1D2V_CP` code. Since the moment we noticed that the transition between radial and orbital motion had nothing to do with the mass of the ion, we have guessed that collisions might be the reason why such a transition is not observed in Argon plasmas. Furthermore, one of the referees of the paper published in *PSST*, encouraged us to take account for collisions and to report the results obtained. This highlights the importance that, finding a model which properly describes the behaviour of ions, has among the plasma diagnosing community. For this reason this is the most timely improvement that we would like to implement in our simulation.
- The introduction of ionisation processes in the `CUPIC1D1V_PP` code. Since this code does not take into account any presheath mechanism, as it has been previously stated, the results of the simulation are in accordance to the sheath solution introduced in Chapter 1, instead of the complete solution. However, in order to improve the results obtained, the simulation should provide results compatible with the complete solution. In this sense, the inclusion of ionisation processes, might help to solve the problems we were having when comparing the results of the modified version of `CUPIC1D1V_PP` with two electron populations with those provided by the model developed by Prof. Tomaž Gyergyek.
- The introduction of multiple species in both simulations. In this sense, the introduction of hot electrons in the cylindrical probe simulation, will have direct applications to the experimental study that is being currently developed in our group with respect the diagnose of bi-maxwelliam plasmas with Langmuir probes. Also, to consider a population of negative ions will be interesting for the study of the potential oscillations predicted by fluid models.
- The introduction of different processes at the probe surface. To consider these processes will represent a step towards the realism of the results provided by our simulations. Particularly important are the processes of secondary electron emission, as well as ion reflection, both of which are relatively common when high negative biasing potentials are chosen for the probe.

Obviously there exist many more processes and ideas that could be implemented into our codes: external magnetic fields, exotic probe geometries and configurations, plasma chemistry processes, etc. However we have mentioned the ones that are easily implemented and affordable with nowadays hardware (it has to be noticed that the more processes included the more computational resources are needed).

Also, the mentioned improvements of the simulations should quickly report interesting results highly demanded by the plasma diagnosing community.

6.5. Conclusions

In this chapter we have summarised the main tasks developed during this research. Also, the main results derived from it have been outlined. Then, the future perspectives and ideas to continue the research that we have just started have been also enumerated. During this last part, special emphasis has been considered when talking about the aspects that more timely need to be addressed.

Finally, I started this chapter by stating the fact that it is not easy to summarise all that has been done and learned during more than four years. For the same reasons, it is not easier to bring to a close this thesis. The last remark that I would like to make is that, all the problems encountered during this research, most of which have been tackled and successfully solved, as well as the experience accumulated, have allowed me to grow as a physicist and a scientist. For this reason, at the end of this stage, I think I have developed the necessary skills to endeavour my research career.

Part IV

Appendixes

Appendix A

Fluid approximation in plasmas: Boltzmann equation and its first moments.

Even though one of the aims of this work has been to showcase the benefits of studying a plasma as a particle system and through simulations, most of the theoretical models used in the study of the contact of a plasma with a Langmuir probe rely on a fluid approximation of the species present in the plasma. As a proof of it, all the models that have been presented in Chapters 1 and 2 rely on fluid approximations to describe ions and electrons. Here we are going to explain some of the equations that have been taken for granted during those chapters.

A.1. Boltzmann transport equation

The Boltzmann transport equation, more commonly known as Boltzmann equation simply, is the equation that describes the statistical behaviour of a thermodynamic system whether it is or not in thermodynamic equilibrium. The equation was named after Ludwig Boltzmann, since he devised it in 1872. It is written in terms of the distribution function of the particles composing the system, $f(\vec{r}, \vec{v})$, which can be defined as the number of particles in a differential volume $d^3\vec{r}\ d^3\vec{v}$ in the phase space $\vec{r}$, $\vec{v}$. The most general form of the equation is written as:

$$\frac{\partial f}{\partial t} = \left(\frac{\partial f}{\partial t}\right)_{\text{force}} + \left(\frac{\partial f}{\partial t}\right)_{\text{diffusion}} + \left(\frac{\partial f}{\partial t}\right)_{\text{collisions}} \tag{A.1}$$

In view of Eq. (A.1), the meaning of the Boltzmann equation becomes pretty obvious. The temporal evolution of the distribution function can be decomposed into the change due to external forces acting on the particles of the system, the change due to the diffusion of particles through the system, and the change due to particle collisions within the system. After some calculations, with the help of Hamilton's equation and Liouville's theorem, Eq. (A.1) can be transformed into the commonly known version of the Boltzmann equation:

$$\frac{df}{dt} = \frac{\partial f}{\partial t} + \underbrace{\vec{v} \cdot \vec{\nabla} f}_{\text{diffusion}} + \underbrace{\frac{\vec{F}}{m} \cdot \frac{\partial f}{\partial \vec{v}}}_{\text{force}} = \left(\frac{\partial f}{\partial t}\right)_{\text{collisions}} \tag{A.2}$$

where m is the mass of the particles in the system and $\vec{F}$ is the force field acting on the particles of the system.

Now, we are going to use the aforementioned Boltzmann equation in order to study our problem, which is the steady flow, $\partial f / \partial t = 0$, of positive ions, $m = m_i$, moving towards the surface of a Langmuir probe. The external force that ions suffer is the electrostatic force, so, $\vec{F} = -e\vec{\nabla}\varphi$. All these considerations lead

us to the following Boltzmann equation:

$$\vec{v} \cdot \vec{\nabla} f - \frac{e}{m_i} \left(\vec{\nabla} \varphi(\vec{r}) \right) \cdot \frac{\partial f}{\partial \vec{v}} = \left(\frac{\partial f}{\partial t} \right)_{\text{collisions}} \tag{A.3}$$

In particular we are going to consider the planar case developed in Chapter 1, as it is the only case that needs considering ionisation (due to electron-neutral collisions) as the presheath mechanism. Nevertheless, the reasonings that we are going to use are applicable to the cylindric or spherical case. So, as in the planar case the problem is monodimensional, Eq. (A.3) is written as:

$$v \frac{\partial f(x,v)}{\partial x} - \frac{e}{m_i} \frac{d\varphi(x)}{dx} \frac{\partial f(x,v)}{\partial v} = \left(\frac{\partial f(x,v)}{\partial t} \right)_{\text{collisions}} \tag{A.4}$$

Last but not least, we have to take care of the collision term in Eq. (A.4). Ionisation is taken into account in the collision term, as it is due to binary collision between electrons and neutral atoms present in the plasma. So, the right hand side of Eq. (A.4) can be written as:

$$\left(\frac{\partial f(x,v)}{\partial t} \right)_{\text{collisions}} = Z n_e(x) f_{\text{new}}(v) \tag{A.5}$$

Z being the ionisation collision frequency, $n_e(x)$ the electron density at x and $f_{\text{new}}(v)$ the velocity distribution function of the new ions created by ionisation. So, finally the Boltzmann equation for our problem would be:

$$v \frac{\partial f(x,v)}{\partial x} - \frac{e}{m_i} \frac{d\varphi(x)}{dx} \frac{\partial f(x,v)}{\partial v} = Z n_e(x) f_{\text{new}}(v) \tag{A.6}$$

There are a few options when it comes to choose the velocity distribution function of the new ions. Many models and texts assume newly created ions to be at rest, so their velocity distribution function would be determined by a delta function centred at zero, $f_{\text{new}}(v) = \delta(v)$. The problem of this assumption is that, as we will see, it leads to wrong results in the limiting case of low ionisation, $Z \to 0$. On the other hand, newly created ions could be assumed to be in equilibrium with the ions coming from the plasma, so both would have the same velocity distribution function, $f_{\text{new}}(v) = f(x,v)/n_i(x)$. Where $n_i(x)$ is the ion density at x. This assumption, even it may looks unreasonable, leads to right results in the case of low ionisation. This statements will become clear in the next section, where we will find the firsts moments of Eq. (A.6).

$$f_{\text{new}} = \begin{cases} \delta(v) & \text{(A.7a)} \\ \dfrac{f(x,v)}{n_i(x)} & \text{(A.7b)} \end{cases}$$

A.2. First and second moments of the Boltzmann equation

Instead of working directly with the Boltzmann equation (A.6), it is usual to use a simplified approach by working with its moments, as by doing so there is no need to work with the distribution function of ions but with their density and velocity fields, $n_i(x)$ and $v_i(x)$. The integrations needed to obtain the different moments of the Boltzmann equation are more or less straightforward, however further details of the specific integration process can be found in chapter 7 of reference [57].

The first moment of the Boltzmann equation is obtained through direct integration in the velocity space of Eq. (A.6). No matter which velocity distribution we choose for the ions created through ionisation, Eq. (A.7a) or Eq. (A.7b), the first moment of the Boltzmann equation yields the same result:

$$\frac{dn_i(x)\overline{v}}{dx} = \frac{dn_i(x)v_i(x)}{dx} = Z n_e(x) \tag{A.8}$$

where $\bar{v}$ is the mean value of the velocity, which is equal to the drift velocity of the ion fluid $v_i(x)$. The velocity of an ion could be written as $v = v_i(x) + v_{th}$, v_{th} being the thermal velocity. As $v_i(x)$ is already an averaged value, we have that $\overline{v_i(x) + v_{th}} = v_i(x) + \overline{v_{th}} = v_i(x)$, since $\overline{v_{th}} = 0$. Usually Eq. (A.8) is

referred to as the **continuity equation**. Let us notice that Eq. (A.8) is the same as Eq. (1.17), except for the sign criterion of the ion velocity.

The second moment of the Boltzmann equation, usually called **balance momentum equation**, is obtained by multiplying Eq. (A.6) by v and then integrating it over the velocity space. Contrary to the previous result, the second moment yields different results when Eq. (A.7a) or Eq. (A.7b) are considered. In the first place, if we consider Eq. (A.7a), the following first moment of the Boltzmann equation is obtained:

$$\frac{dn_i(x)\overline{vv}}{dx} + \frac{e}{m_i} n_i(x) \frac{d\varphi(x)}{dx} = 0 \Rightarrow$$

$$\Rightarrow \frac{dn_i(x)v_i^2(x)}{dx} + \frac{d\cancelto{p_i(x)/m_i}{n_i(x)\overline{v_{th}^2}}}{dx} + \frac{d2n_i(x)v_i(x)\cancelto{0}{\overline{v_{th}}}}{dx} + \frac{e}{m_i} n_i(x) \frac{d\varphi(x)}{dx} = 0 \Rightarrow$$

$$\frac{dn_i(x)v_i^2(x)}{dx} + \frac{1}{m_i}\frac{dp_i(x)}{dx} + \frac{e}{m_i} n_i(x) \frac{d\varphi(x)}{dx} = 0 \tag{A.9}$$

Where $p_i(x)$ is the partial pressure of the ion fluid. The term $dp_i(x)/dx$, or in general $\vec{\nabla} \cdot \mathbf{P}$, $\mathbf{P}$ being the stress tensor, is the diffusion term in the moment balance equation. We have to notice that this term is due to the thermal motion of ions, so, in case the ions are considered monoenergetic, it can be neglected. From now on, we are going to neglect this term, since here we are developing the planar case of chapter 1, where ions where considered to be monoenergetic. However, in chapter 2 we considered this term when reviewing Fernández Palop's model in section 2.4.2.

Now, after neglecting the diffusion term, if we take Eq. (A.8) into Eq. (A.9):

$$n_i(x)v_i(x)\frac{dv_i(x)}{dx} + v_i(x)Zn_e(x) + \frac{e}{m_i} n_i(x) \frac{d\varphi(x)}{dx} = 0 \tag{A.10}$$

The second term in Eq. (A.10) is usually called "ionisation drag force". Its name is due to the fact that ions are created at rest and, consequently, the flow velocity of ions is decreased with respect to the velocity that ions coming from the plasma have. This term was diminished in Section 1.4 arguing that in the case of low ionisation the drag force does not contribute significantly to the dynamic of the motion of ions. So, by neglecting the drag force term, Eq. (A.10) becomes Eq. (1.15), which is the balance momentum equation considered in Section 1.4. Nevertheless we are going to see how Eq. (A.9) yields wrong results in the limiting case $Z \to 0$.

Eq. (A.9) can be written as:

$$\frac{dn_i(x)v_i^2(x)}{d\varphi(x)}\frac{d\varphi(x)}{dx} + \frac{e}{m_i} n_i(x) \frac{d\varphi(x)}{dx} = 0 \Rightarrow \frac{dn_i(\varphi)v_i^2(\varphi)}{d\varphi} + \frac{e}{m_i} n_i(\varphi) = 0 \tag{A.11}$$

Also, we know that the quasineutral solution is given by:

$$n_i(\varphi) = n_e(\varphi) = n_{e0} \exp\left(\frac{e\varphi}{k_B T_e}\right) \tag{A.12}$$

Now, if we take Eq. (A.12) into Eq. (A.11) and solve for $v_i(\varphi)$, we can obtain the following expression:

$$v_i(\varphi) = \frac{1}{\sqrt{m_i}} \left[\frac{k_B T_e \left(1 - \exp\left(\frac{e\varphi}{k_B T_e}\right)\right)}{\exp\left(\frac{e\varphi}{k_B T_e}\right)} \right]^{1/2} \tag{A.13}$$

Eq. (A.13) is independent of the ionisation rate, so it must be valid for any Z value, in particular it should be correct for $Z = 0$. But, for the case of no ionisation, we know that the energy conservation law yields:

$$v_i(\varphi) = \sqrt{-\frac{2e\varphi}{m_i}} \tag{A.14}$$

So, because of the contradictory results given by Eq. (A.14) and Eq. (A.13) we can say that Eq. (A.9) is not correct for the case of low ionisation, and so Eq. (A.10) does.

Now, let us obtain the second moment of the Boltzmann equation considering that the new ions created by ionisation are in equilibrium with ions coming from the plasma. So, by considering Eq. (A.7b), multiplying Eq. (A.6) by v and integrating in the velocity space, we obtain the following expression:

$$\frac{dn_i(x)v_i^2(x)}{dx} + \frac{e}{m_i}n_i(x)\frac{d\varphi(x)}{dx} = Zn_e(x)v_i(x) \tag{A.15}$$

where we have again neglected the diffusion term. And if we take Eq. (A.8) into Eq. (A.15):

$$n_i(x)v_i(x)\frac{dv_i(x)}{dx} + \frac{e}{m_i}n_i(x)\frac{d\varphi(x)}{dx} = 0 \tag{A.16}$$

We can see that Eq. (A.16) is equal to Eq. (1.15), which is the equation that leads to correct results in the limit of low ionisation. When the second term in Eq. (A.10) is neglected we are artificially using the right moment balance equation. Nevertheless, there is no need of neglecting the ionisation drag force term if we assume that newly created ions are in equilibrium with ions coming from the plasma, as we have seen. The problem of using Eq. (A.7a) is that we are trying to describe with the same velocity field ions that comes from the plasma with a certain velocity and ions that are created along the sheath at rest.

We have to remark here that, independently of the assumptions considered, the proper balance momentum equation that should be used for the case of low ionisation is given by Eq. (A.16). We also have to notice that these results can be also obtained in the case of cylindrical or spherical probes. Actually in those cases, if ionisation is not taken into account, the collision term in Eq. (A.4) is neglected, and calculations are straightforward.

Appendix B

CUPIC1D1V_PP sources

This appendix is devoted to the source code of our simulation of the contact of a planar Langmuir probe with a plasma. The code is divided into seven modules, each one taking care of an specific task. Also, there are a few extra header files that are loaded from the previous modules whenever they are needed, and a makefile that takes care of the compilation of the different modules to produce the simulation binary (makefile).

In the following sections the source files of the different modules are shown.

B.1. Main module

This is the main module of the simulation, it handles the simulation by calling functions that belongs to the rest of the modules. (sources: main.cu)

```
/****************************************************************************************************
 *                                                                                                  *
 *    This file is part of CUPIC1D1V_PP, a code that simulates the interaction between a plasma and *
 *    a planar Langmuir probe in 1D using PIC techniques accelerated with the use of GPU hardware   *
 *    (CUDA, extension of C/C++)                                                                    *
 *                                                                                                  *
 ****************************************************************************************************/

/********************************************* HEADERS *********************************************/

#include "stdh.h"
#include "init.h"
#include "cc.h"
#include "mesh.h"
#include "particles.h"
#include "diagnostic.h"

/****************************************** MAIN FUNCTION ******************************************/

int main (int argc, const char* argv[])
{
  /*------------------------------------- function variables --------------------------------------*/
  // host variables definition
  const double dt = init_dt();           // time step
  const int n_ini = init_n_ini();        // number of first iteration
  const int n_prev = init_n_prev();      // number of iterations before start analizing
  const int n_save = init_n_save();      // number of iterations between diagnostics
  const int n_fin = init_n_fin();        // number of last iteration
  const int nn = init_nn();              // number of nodes

  double t;                              // time of simulation
  int num_e, num_i;                      // number of particles (electrons and ions)
  double U_e, U_i;                       // system energy for electrons and ions
  double mi = init_mi();                 // ion mass
  double dtin_e = init_dtin_e();         // time between electron insertions
  double dtin_i = init_dtin_i();         // time between ion insertions
  double vd_e = init_vd_e();             // drift velocity of electrons
  double vd_i = init_vd_i();             // drift velocity of ions
  double q_p = 0;                        // probe's acumulated charge
  char filename[50];                     // filename for saved data

  ifstream ifile;
  ofstream ofile;
```

```
44
45    // device variables definition
46    double *d_rho, *d_phi, *d_E;            // mesh properties
47    double *d_avg_rho, *d_avg_phi, *d_avg_E;  // mesh averaged properties
48    double *d_avg_ddf_e, *d_avg_vdf_e;      // density and velocity distribution function for electrons
49    double v_max_e = init_v_max_e();         // maximun velocity of electrons (for histograms)
50    double v_min_e = init_v_min_e();         // minimun velocity of electrons (for histograms)
51    double *d_avg_ddf_i, *d_avg_vdf_i;       // density and velocity distribution function for ions
52    double v_max_i = init_v_max_i();         // maximun velocity of ions (for histograms)
53    double v_min_i = init_v_min_i();         // minimun velocity of ions (for histograms)
54    int count_df_e = 0;                      // |
55    int count_df_i = 0;                      // |
56    int count_rho = 0;                       // |-> counters for avg data
57    int count_phi = 0;                       // |
58    int count_E = 0;                         // |
59    particle *d_e, *d_i;                     // particles vectors
60    curandStatePhilox4_32_10_t *state;       // philox state for __device__ random number generation
61
62    /*---------------------------------------- function body -----------------------------------------*/
63
64    //---- INITIALITATION OF SIMULATION
65
66    // initialize device and simulation variables
67    init_dev();
68    init_sim(&d_rho, &d_phi, &d_E, &d_avg_rho, &d_avg_phi, &d_avg_E, &d_e, &num_e, &d_i, &num_i,
69             &d_avg_ddf_e, &d_avg_vdf_e, &d_avg_ddf_i, &d_avg_vdf_i, &t, &state);
70
71    // save initial state
72    sprintf(filename, "../output/particles/electrons_t_%d", n_ini);
73    particles_snapshot(d_e, num_e, filename);
74    sprintf(filename, "../output/particles/ions_t_%d", n_ini);
75    particles_snapshot(d_i, num_i, filename);
76    sprintf(filename, "../output/charge/avg_charge_t_%d", n_ini);
77    save_mesh(d_avg_rho, filename);
78    sprintf(filename, "../output/potential/avg_potential_t_%d", n_ini);
79    save_mesh(d_avg_phi, filename);
80    sprintf(filename, "../output/field/avg_field_t_%d", n_ini);
81    save_mesh(d_avg_E, filename);
82    t += dt;
83
84    //---- SIMULATION BODY
85
86    for (int i = n_ini+1; i <= n_fin; i++, t += dt) {
87      // simulate one time step
88      charge_deposition(d_rho, d_e, num_e, d_i, num_i);
89      poisson_solver(1.0e-4, d_rho, d_phi);
90      field_solver(d_phi, d_E);
91      particle_mover(d_e, num_e, d_i, num_i, d_E);
92      cc(t, &num_e, &d_e, &dtin_e, &vd_e, &num_i, &d_i, &dtin_i, &vd_i, &q_p, d_phi, d_E, state);
93
94      // average mesh variables and distribution functions
95      avg_mesh(d_rho, d_avg_rho, &count_rho);
96      avg_mesh(d_phi, d_avg_phi, &count_phi);
97      avg_mesh(d_E, d_avg_E, &count_E);
98      eval_df(d_avg_ddf_e, d_avg_vdf_e, v_max_e, v_min_e, d_e, num_e, &count_df_e);
99      eval_df(d_avg_ddf_i, d_avg_vdf_i, v_max_i, v_min_i, d_i, num_i, &count_df_i);
100
101     // store data
102     if (i>=n_prev && i%n_save==0) {
103       // save particles (snapshot)
104       sprintf(filename, "../output/particles/electrons_t_%d", i);
105       particles_snapshot(d_e, num_e, filename);
106       sprintf(filename, "../output/particles/ions_t_%d", i);
107       particles_snapshot(d_i, num_i, filename);
108
109       // save mesh properties
110       sprintf(filename, "../output/charge/avg_charge_t_%d", i);
111       save_mesh(d_avg_rho, filename);
112       sprintf(filename, "../output/potential/avg_potential_t_%d", i);
113       save_mesh(d_avg_phi, filename);
114       sprintf(filename, "../output/field/avg_field_t_%d", i);
115       save_mesh(d_avg_E, filename);
116
117       // save distribution functions
118       sprintf(filename, "../output/particles/electrons_ddf_t_%d", i);
119       save_ddf(d_avg_ddf_e, filename);
120       sprintf(filename, "../output/particles/ions_ddf_t_%d", i);
121       save_ddf(d_avg_ddf_i, filename);
122       sprintf(filename, "../output/particles/electrons_vdf_t_%d", i);
123       save_vdf(d_avg_vdf_e, v_max_e, v_min_e, filename);
124       sprintf(filename, "../output/particles/ions_vdf_t_%d", i);
125       save_vdf(d_avg_vdf_i, v_max_i, v_min_i, filename);
126
127       // save log
128       U_e = eval_particle_energy(d_phi,  d_e, 1.0, -1.0, num_e);
129       U_i = eval_particle_energy(d_phi,  d_i, mi, 1.0, num_i);
130       save_log(t, num_e, num_i, U_e, U_i, vd_e, vd_i, d_phi);
```

```
131	    }
132	  }
133	
134	  //---- END OF SIMULATION
135	
136	  // update input data file and finish simulation
137	  ifile.open("../input/input_data");
138	  ofile.open("../input/input_data_new");
139	  if (ifile.is_open() && ofile.is_open()) {
140	    ifile.getline(filename, 50);
141	    ofile << filename << endl;
142	    ifile.getline(filename, 50);
143	    ofile << "n_ini␣=␣" << n_fin << ";" << endl;
144	    ifile.getline(filename, 50);
145	    while (!ifile.eof()) {
146	      ofile << filename << endl;
147	      ifile.getline(filename, 50);
148	    }
149	  }
150	  ifile.close();
151	  ofile.close();
152	  system("mv␣../input/input_data_new␣../input/input_data");
153	
154	  cout << "Simulation␣finished!" << endl;
155	  return 0;
156	}
```

Code B.1: CUPIC1D1V_PP source file main.cu

B.2. Initialisation module

This is the module that handles the initialisation of the different variables of the simulation. It also prescribes the initial conditions for the system. (sources: init.cu, init.h)

```
1	/****************************************************************************************************
2	 *                                                                                                  *
3	 *    This file is part of CUPIC1D1V_PP, a code that simulates the interaction between a plasma and  *
4	 *    a planar Langmuir probe in 1D using PIC techniques accelerated with the use of GPU hardware    *
5	 *    (CUDA, extension of C/C++)                                                                    *
6	 *                                                                                                  *
7	 ****************************************************************************************************/
8	
9	/********************************************** HEADERS *******************************************/
10	
11	#include "init.h"
12	
13	/*********************************** HOST FUNCTION DEFINITIONS ***********************************/
14	
15	void init_dev(void)
16	{
17	  /*-------------------------------------- function variables ---------------------------------------*/
18	  // host memory
19	  int dev;
20	  int devcnt;
21	  cudaDeviceProp devProp;
22	  cudaError_t cuError;
23	
24	  // device memory
25	
26	  /*---------------------------------------- function body -----------------------------------------*/
27	
28	  // check for devices instaled in the host
29	  cuError = cudaGetDeviceCount(&devcnt);
30	  if (0 != cuError)
31	  {
32	    printf("Cuda␣error␣(%d)␣detected␣in␣'init_dev(void)'\n", cuError);
33	    cout << "exiting␣simulation..." << endl;
34	    exit(1);
35	  }
36	  cout << devcnt << "␣devices␣present␣in␣the␣host:" << endl;
37	  for (dev = 0; dev < devcnt; dev++)
38	  {
39	    cudaGetDeviceProperties(&devProp, dev);
40	    cout << "␣␣-␣Device␣" << dev << ":" << endl;
41	    cout << "␣␣␣␣#␣" << devProp.name << endl;
42	    cout << "␣␣␣␣#␣Compute␣capability␣" << devProp.major << "." << devProp.minor << endl;
43	  }
44	
45	  // ask wich device to use
46	  cout << "Select␣in␣wich␣device␣simulation␣must␣be␣run:␣0" << endl;
47	  dev = 0;  //cin >> dev;
48	
```

```
  // set device to be used and reset it
  cudaSetDevice(dev);
  cudaDeviceReset();

  return;
}

void init_sim(double **d_rho, double **d_phi, double **d_E, double **d_avg_rho, double **d_avg_phi,
              double **d_avg_E, particle **d_e, int *num_e, particle **d_i, int *num_i,
              double **d_avg_ddf_e, double **d_avg_vdf_e, double **d_avg_ddf_i, double **d_avg_vdf_i,
              double *t, curandStatePhilox4_32_10_t **state)
{
  /*------------------------------------ function variables --------------------------------------*/
  // host memory
  const double dt = init_dt();
  const int n_ini = init_n_ini();

  // device memory

  /*---------------------------------------- function body -----------------------------------------*/

  cout << "n␣=␣" << init_n() << endl;
  // check if simulation start from initial condition or saved state
  if (n_ini == 0) {
    // adjust initial time
    *t = 0.;

    // create particles
    create_particles(d_i, num_i, d_e, num_e, state);

    // initialize mesh variables and their averaged counterparts
    initialize_mesh(d_rho, d_phi, d_E, *d_i, *num_i, *d_e, *num_e);

    // adjust velocities for leap-frog scheme
    adjust_leap_frog(*d_i, *num_i, *d_e, *num_e, *d_E);

    //initialize diagnostic variables
    initialize_avg_mesh(d_avg_rho, d_avg_phi, d_avg_E);
    initialize_avg_df(d_avg_ddf_e, d_avg_vdf_e, d_avg_ddf_i, d_avg_vdf_i);

    cout << "Simulation␣initialized␣with␣" << *num_e*2 << "␣particles." << endl << endl;
  } else if (n_ini > 0) {
    // adjust initial time
    *t = n_ini*dt;

    // read particle from file
    load_particles(d_i, num_i, d_e, num_e, state);

    // initialize mesh variables
    initialize_mesh(d_rho, d_phi, d_E, *d_i, *num_i, *d_e, *num_e);

    //initialize diagnostic variables
    initialize_avg_mesh(d_avg_rho, d_avg_phi, d_avg_E);
    initialize_avg_df(d_avg_ddf_e, d_avg_vdf_e, d_avg_ddf_i, d_avg_vdf_i);

    cout << "Simulation␣state␣loaded␣from␣time␣t␣=␣" << *t << endl;
  } else {
    cout << "Wrong␣input␣parameter␣(n_ini<0)" << endl;
    cout << "Stoppin␣simulation" << endl;
    exit(1);
  }

  return;
}

void create_particles(particle **d_i, int *num_i, particle **d_e, int *num_e,
                      curandStatePhilox4_32_10_t **state)
{
  /*------------------------------------ function variables --------------------------------------*/
  // host memory
  const double n = init_n();          // plasma density
  const double mi = init_mi();        // ion's mass
  const double me = init_me();        // electron's mass
  const double kti = init_kti();      // ion's thermal energy
  const double kte = init_kte();      // electron's thermal energy
  const double L = init_L();          // size of simulation
  const double ds = init_ds();        // spacial step
  cudaError_t cuError;                // cuda error variable

  // device memory

  /*---------------------------------------- function body -----------------------------------------*/

  // initialize curand philox states
  cuError = cudaMalloc ((void **) state, CURAND_BLOCK_DIM*sizeof(curandStatePhilox4_32_10_t));
  cu_check(cuError, __FILE__, __LINE__);
  cudaGetLastError();
```

```
136   init_philox_state<<<1, CURAND_BLOCK_DIM>>>(*state);
137   cu_sync_check(__FILE__, __LINE__);
138 
139   // calculate initial number of particles
140   //*num_i = int(n*ds*ds*L);
141   *num_i = 0;
142   *num_e = *num_i;
143 
144   // allocate device memory for particle vectors
145   cuError = cudaMalloc ((void **) d_i, (*num_i)*sizeof(particle));
146   cu_check(cuError, __FILE__, __LINE__);
147   cuError = cudaMalloc ((void **) d_e, (*num_e)*sizeof(particle));
148   cu_check(cuError, __FILE__, __LINE__);
149 
150   // create particles (electrons)
151   cudaGetLastError();
152   create_particles_kernel<<<1, CURAND_BLOCK_DIM>>>(*d_e, *num_e, kte, me, L, *state);
153   cu_sync_check(__FILE__, __LINE__);
154 
155   // create particles (ions)
156   cudaGetLastError();
157   create_particles_kernel<<<1, CURAND_BLOCK_DIM>>>(*d_i, *num_i, kti, mi, L, *state);
158   cu_sync_check(__FILE__, __LINE__);
159 
160   return;
161 }
162 
163 void initialize_mesh(double **d_rho, double **d_phi, double **d_E, particle *d_i, int num_i,
164                      particle *d_e, int num_e)
165 {
166   /*------------------------------------ function variables --------------------------------------*/
167   // host memory
168   const double phi_p = init_phi_p();                              // probe's potential
169   const double phi_s = -0.5*init_mi()*init_vd_i()*init_vd_i();   // sheath edge potential
170   const int nn = init_nn();                                       // number of nodes
171   const int nc = init_nc();                                       // number of cells
172   double *h_phi;                                                  // host vector for potentials
173   cudaError_t cuError;                                            // cuda error variable
174 
175   // device memory
176 
177   /*-------------------------------------- function body -----------------------------------------*/
178 
179   // allocate host memory for potential
180   h_phi = (double*) malloc(nn*sizeof(double));
181 
182   // allocate device memory for mesh variables
183   cuError = cudaMalloc ((void **) d_rho, nn*sizeof(double));
184   cu_check(cuError, __FILE__, __LINE__);
185   cuError = cudaMalloc ((void **) d_phi, nn*sizeof(double));
186   cu_check(cuError, __FILE__, __LINE__);
187   cuError = cudaMalloc ((void **) d_E, nn*sizeof(double));
188   cu_check(cuError, __FILE__, __LINE__);
189 
190   //initialize potential (host memory)
191   for (int i = 0; i < nn; i++)
192   {
193     h_phi[i] = phi_p + double(i)*(phi_s-phi_p)/double(nc);
194   }
195 
196   // copy potential from host to device memory
197   cuError = cudaMemcpy (*d_phi, h_phi, nn*sizeof(double), cudaMemcpyHostToDevice);
198   cu_check(cuError, __FILE__, __LINE__);
199 
200   // free host memory
201   free(h_phi);
202 
203   // deposit charge into the mesh nodes
204   charge_deposition(*d_rho, d_e, num_e, d_i, num_i);
205 
206   // solve poisson equation
207   poisson_solver(1.0e-4, *d_rho, *d_phi);
208 
209   // derive electric fields from potential
210   field_solver(*d_phi, *d_E);
211 
212   return;
213 }
214 
215 void initialize_avg_mesh(double **d_avg_rho, double **d_avg_phi, double **d_avg_E)
216 {
217   /*------------------------------------ function variables --------------------------------------*/
218   // host memory
219   const int nn = init_nn();   // number of nodes
220   cudaError_t cuError;        // cuda error variable
221 
222   // device memory
```

```
223
224   /*--------------------------------------- function body ------------------------------------------*/
225
226   // allocate device memory for averaged mesh variables
227   cuError = cudaMalloc ((void **) d_avg_rho, nn*sizeof(double));
228   cu_check(cuError, __FILE__, __LINE__);
229   cuError = cudaMalloc ((void **) d_avg_phi, nn*sizeof(double));
230   cu_check(cuError, __FILE__, __LINE__);
231   cuError = cudaMalloc ((void **) d_avg_E, nn*sizeof(double));
232   cu_check(cuError, __FILE__, __LINE__);
233
234   // initialize to zero averaged variables
235   cuError = cudaMemset ((void *) *d_avg_rho, 0, nn*sizeof(double));
236   cu_check(cuError, __FILE__, __LINE__);
237   cuError = cudaMemset ((void *) *d_avg_phi, 0, nn*sizeof(double));
238   cu_check(cuError, __FILE__, __LINE__);
239   cuError = cudaMemset ((void *) *d_avg_E, 0, nn*sizeof(double));
240   cu_check(cuError, __FILE__, __LINE__);
241
242   return;
243 }
244
245 void initialize_avg_df(double **d_avg_ddf_e, double **d_avg_vdf_e, double **d_avg_ddf_i,
246                        double **d_avg_vdf_i)
247 {
248   /*------------------------------------- function variables ---------------------------------------*/
249   // host memory
250   const int n_bin_ddf = init_n_bin_ddf();   // number of bins for density distribution function
251   const int n_bin_vdf = init_n_bin_vdf();   // number of bins for velocity distribution function
252   const int n_vdf = init_n_vdf();           // number of velocity distribution functions to calculate
253   cudaError_t cuError;        // cuda error variable
254
255   // device memory
256
257   /*--------------------------------------- function body ------------------------------------------*/
258
259   // allocate device memory for averaged distribution functions
260   cuError = cudaMalloc ((void **) d_avg_ddf_e, n_bin_ddf*sizeof(double));
261   cu_check(cuError, __FILE__, __LINE__);
262   cuError = cudaMalloc ((void **) d_avg_ddf_i, n_bin_ddf*sizeof(double));
263   cu_check(cuError, __FILE__, __LINE__);
264   cuError = cudaMalloc ((void **) d_avg_vdf_e, n_bin_vdf*n_vdf*sizeof(double));
265   cu_check(cuError, __FILE__, __LINE__);
266   cuError = cudaMalloc ((void **) d_avg_vdf_i, n_bin_vdf*n_vdf*sizeof(double));
267   cu_check(cuError, __FILE__, __LINE__);
268
269   // initialize to zero averaged distribution functions
270   cuError = cudaMemset ((void *) *d_avg_ddf_e, 0, n_bin_ddf*sizeof(double));
271   cu_check(cuError, __FILE__, __LINE__);
272   cuError = cudaMemset ((void *) *d_avg_ddf_i, 0, n_bin_ddf*sizeof(double));
273   cu_check(cuError, __FILE__, __LINE__);
274   cuError = cudaMemset ((void *) *d_avg_vdf_e, 0, n_bin_vdf*n_vdf*sizeof(double));
275   cu_check(cuError, __FILE__, __LINE__);
276   cuError = cudaMemset ((void *) *d_avg_vdf_i, 0, n_bin_vdf*n_vdf*sizeof(double));
277   cu_check(cuError, __FILE__, __LINE__);
278
279   return;
280 }
281
282 void adjust_leap_frog(particle *d_i, int num_i, particle *d_e, int num_e, double *d_E)
283 {
284   /*------------------------------------- function variables ---------------------------------------*/
285   // host memory
286   const double mi = init_mi();          // ion's mass
287   const double me = init_me();          // electron's mass
288   const double ds = init_ds();          // spatial step size
289   const double dt = init_dt();          // temporal step size
290   const int nn = init_nn();             // number of nodes
291
292   dim3 griddim, blockdim;               // kernel execution configurations
293   size_t sh_mem_size;                   // shared memory size
294
295   // device memory
296
297   /*--------------------------------------- function body ------------------------------------------*/
298
299   // set grid and block dimensions for fix_velocity kernel
300   griddim = 1;
301   blockdim = PAR_MOV_BLOCK_DIM;
302
303   // set shared memory size for fix_velocity kernel
304   sh_mem_size = nn*sizeof(double);
305
306   // fix velocities (electrons)
307   cudaGetLastError();
308   fix_velocity<<<griddim, blockdim, sh_mem_size>>>(-1.0, me, num_e, d_e, dt, ds, nn, d_E);
309   cu_sync_check(__FILE__, __LINE__);
```

```
310
311   // fix velocities (ions)
312   cudaGetLastError();
313   fix_velocity<<<griddim, blockdim, sh_mem_size>>>(1.0, mi, num_i, d_i, dt, ds, nn, d_E);
314   cu_sync_check(__FILE__, __LINE__);
315
316   return;
317 }
318
319 void load_particles(particle **d_i, int *num_i, particle **d_e, int *num_e,
320                     curandStatePhilox4_32_10_t **state)
321 {
322   /*------------------------------------ function variables ---------------------------------------*/
323   // host memory
324   char filename[50];
325   cudaError_t cuError;              // cuda error variable
326
327   // device memory
328
329   /*---------------------------------------- function body -----------------------------------------*/
330
331   // initialize curand philox states
332   cuError = cudaMalloc ((void **) state, CURAND_BLOCK_DIM*sizeof(curandStatePhilox4_32_10_t));
333   cu_check(cuError, __FILE__, __LINE__);
334   cudaGetLastError();
335   init_philox_state<<<1, CURAND_BLOCK_DIM>>>(*state);
336   cu_sync_check(__FILE__, __LINE__);
337
338   // load particles
339   sprintf(filename, "./ions.dat");
340   read_particle_file(filename, d_i, num_i);
341   sprintf(filename, "./electrons.dat");
342   read_particle_file(filename, d_e, num_e);
343
344   return;
345 }
346
347 void read_particle_file(string filename, particle **d_p, int *num_p)
348 {
349   /*------------------------------------ function variables ---------------------------------------*/
350   // host memory
351   particle *h_p;                 // host vector for particles
352   ifstream myfile;               // file variables
353   char line[150];
354   cudaError_t cuError;           // cuda error variable
355
356   // device memory
357
358   /*---------------------------------------- function body -----------------------------------------*/
359
360   // get number of particles (test if n is correctly evaluated)
361   *num_p = 0;
362   myfile.open(filename.c_str());
363   if (myfile.is_open()) {
364     myfile.getline(line, 150);
365     while (!myfile.eof()) {
366       myfile.getline(line, 150);
367       *num_p += 1;
368     }
369   } else {
370     cout << "Error.␣Can't␣open␣" << filename << "␣file" << endl;
371   }
372   myfile.close();
373
374   // allocate host and device memory for particles
375   h_p = (particle*) malloc(*num_p*sizeof(particle));
376   cuError = cudaMalloc ((void **) d_p, *num_p*sizeof(particle));
377   cu_check(cuError, __FILE__, __LINE__);
378
379   // read particles from file and store in host memory
380   myfile.open(filename.c_str());
381   if (myfile.is_open()) {
382     myfile.getline(line, 150);
383     for (int i = 0; i<*num_p; i++) {
384       myfile.getline(line, 150);
385       sscanf (line, "␣%le␣%le␣\n", &h_p[i].r, &h_p[i].v);
386     }
387   } else {
388     cout << "Error.␣Can't␣open␣" << filename << "␣file" << endl;
389   }
390   myfile.close();
391
392   // copy particle vector from host to device memory
393   cuError = cudaMemcpy (*d_p, h_p, *num_p*sizeof(particle), cudaMemcpyHostToDevice);
394   cu_check(cuError, __FILE__, __LINE__);
395
396   // free host memory
```

```
397   free(h_p);
398
399   return;
400 }
401
402 template <typename type> void read_input_file(type *data, int n)
403 {
404   /*------------------------------------- function variables --------------------------------------*/
405   ifstream myfile;
406   char line[80];
407
408   /*--------------------------------------- function body -----------------------------------------*/
409   myfile.open("../input/input_data");
410   if (myfile.is_open()) {
411     for (int i = 0; i < n; i++) myfile.getline(line, 80);
412     if (sizeof(type) == sizeof(int)) {
413       sscanf (line, "%*s␣=␣%d;\n", (int*) data);
414     } else if (sizeof(type) == sizeof(double)) {
415       sscanf (line, "%*s␣=␣%lf;\n", (double*) data);
416     }
417   } else {
418     cout << "Error.␣Input␣data␣file␣could␣not␣be␣opened" << endl;
419     exit(1);
420   }
421   myfile.close();
422   return;
423 }
424
425 double init_qi(void)
426 {
427   /*------------------------------------- function variables --------------------------------------*/
428
429   /*--------------------------------------- function body -----------------------------------------*/
430
431   return 1.0;
432 }
433
434 double init_qe(void)
435 {
436   /*------------------------------------- function variables --------------------------------------*/
437
438   /*--------------------------------------- function body -----------------------------------------*/
439
440   return -1.0;
441 }
442
443 double init_mi(void)
444 {
445   /*------------------------------------- function variables --------------------------------------*/
446   static double gamma = 0.0;
447
448   /*--------------------------------------- function body -----------------------------------------*/
449
450   if (gamma == 0.0) read_input_file(&gamma, 12);
451   return gamma;
452 }
453
454 double init_me(void)
455 {
456   /*------------------------------------- function variables --------------------------------------*/
457
458   /*--------------------------------------- function body -----------------------------------------*/
459
460   return 1.0;
461 }
462
463 double init_kti(void)
464 {
465   /*------------------------------------- function variables --------------------------------------*/
466   static double beta = 0.0;
467
468   /*--------------------------------------- function body -----------------------------------------*/
469
470   if (beta == 0.0) read_input_file(&beta, 9);
471   return beta;
472 }
473
474 double init_kte(void)
475 {
476   /*------------------------------------- function variables --------------------------------------*/
477
478   /*--------------------------------------- function body -----------------------------------------*/
479
480   return 1.0;
481 }
482
483 double init_vd_i(void)
```

```
{
  /*--------------------------------------- function variables ----------------------------------------*/
  static double vd_i = -10.0;

  /*----------------------------------------- function body ------------------------------------------*/

  if (vd_i == -10.0) read_input_file(&vd_i, 11);
  return vd_i;
}

double init_vd_e(void)
{
  /*--------------------------------------- function variables ----------------------------------------*/
  static double vd_e = -10.0;

  /*----------------------------------------- function body ------------------------------------------*/

  if (vd_e == -10.0) read_input_file(&vd_e, 10);
  return vd_e;
}

double init_phi_p(void)
{
  /*--------------------------------------- function variables ----------------------------------------*/
  static double phi_p = 0.0;

  /*----------------------------------------- function body ------------------------------------------*/

  if (phi_p == 0.0) read_input_file(&phi_p, 14);
  return phi_p;
}

double init_n(void)
{
  /*--------------------------------------- function variables ----------------------------------------*/
  const double Dl = init_Dl();
  static double n = 0.0;

  /*----------------------------------------- function body ------------------------------------------*/

  if (n == 0.0) {
    read_input_file(&n, 7);
    n *= Dl*Dl*Dl;
  }
  return n;
}

double init_L(void)
{
  /*--------------------------------------- function variables ----------------------------------------*/
  static double L = init_ds() * (double) init_nc();

  /*----------------------------------------- function body ------------------------------------------*/

  return L;
}

double init_ds(void)
{
  /*--------------------------------------- function variables ----------------------------------------*/
  static double ds = 0.0;

  /*----------------------------------------- function body ------------------------------------------*/

  if (ds == 0.0) read_input_file(&ds, 17);
  return ds;
}

double init_dt(void)
{
  /*--------------------------------------- function variables ----------------------------------------*/
  static double dt = 0.0;

  /*----------------------------------------- function body ------------------------------------------*/

  if (dt == 0.0) read_input_file(&dt, 18);
  return dt;
}

double init_epsilon0(void)
{
  /*--------------------------------------- function variables ----------------------------------------*/
  double Te;
  const double Dl = init_Dl();
  static double epsilon0 = 0.0;
  /*----------------------------------------- function body ------------------------------------------*/

```

```
571    if (epsilon0 == 0.0) {
572      read_input_file(&Te, 8);
573      epsilon0 = CST_EPSILON;                        // SI units
574      epsilon0 /= pow(Dl*sqrt(CST_ME/(CST_KB*Te)),2); // time units
575      epsilon0 /= CST_E*CST_E;                       // charge units
576      epsilon0 *= Dl*Dl*Dl;                          // length units
577      epsilon0 *= CST_ME;                            // mass units
578    }
579    return epsilon0;
580  }
581
582  int init_nc(void)
583  {
584    /*------------------------------------- function variables -------------------------------------*/
585    static int nc = 0;
586
587    /*---------------------------------------- function body ----------------------------------------*/
588
589    if (nc == 0) read_input_file(&nc, 16);
590    return nc;
591  }
592
593  int init_nn(void)
594  {
595    /*------------------------------------- function variables -------------------------------------*/
596    static int nn = init_nc()+1;
597
598    /*---------------------------------------- function body ----------------------------------------*/
599
600    return nn;
601  }
602
603  double init_dtin_i(void)
604  {
605    /*------------------------------------- function variables -------------------------------------*/
606    const double n = init_n();
607    const double ds = init_ds();
608    const double mi = init_mi();
609    const double kti = init_kti();
610    const double vd_i = init_vd_i();
611    const double phi_s = -0.5*init_mi()*init_vd_i()*init_vd_i();
612    const double phi_p = init_phi_p();
613    static double dtin_i = 0.0;
614
615    /*---------------------------------------- function body ----------------------------------------*/
616
617    if (dtin_i == 0.0) {
618      dtin_i = n*sqrt(kti/(2.0*PI*mi))*exp(-0.5*mi*vd_i*vd_i/kti);  // thermal component of input flux
619      dtin_i -= 0.5*n*vd_i*(1.0+erf(sqrt(0.5*mi/kti)*(-vd_i)));     // drift component of input flux
620      dtin_i *= exp(phi_s)*0.5*(1.0+erf(sqrt(phi_s-phi_p)));        // correction on density at sheath edge
621      dtin_i *= ds*ds;      // number of particles that enter the simulation per unit of time
622      dtin_i = 1.0/dtin_i;  // time between consecutive particles injection
623    }
624    return dtin_i;
625  }
626
627  double init_dtin_e(void)
628  {
629    /*------------------------------------- function variables -------------------------------------*/
630    const double n = init_n();
631    const double ds = init_ds();
632    const double me = init_me();
633    const double kte = init_kte();
634    const double vd_e = init_vd_e();
635    const double phi_s = -0.5*init_mi()*init_vd_i()*init_vd_i();
636    const double phi_p = init_phi_p();
637    static double dtin_e = 0.0;
638
639    /*---------------------------------------- function body ----------------------------------------*/
640
641    if (dtin_e == 0.0) {
642      dtin_e = n*sqrt(kte/(2.0*PI*me))*exp(-0.5*me*vd_e*vd_e/kte);  // thermal component of input flux
643      dtin_e -= 0.5*n*vd_e*(1.0+erf(sqrt(0.5*me/kte)*(-vd_e)));     // drift component of input flux
644      dtin_e *= exp(phi_s)*0.5*(1.0+erf(sqrt(phi_s-phi_p)));        // correction on density at sheath edge
645      dtin_e *= ds*ds;      // number of particles that enter the simulation per unit of time
646      dtin_e = 1.0/dtin_e;  // time between consecutive particles injection
647    }
648    return dtin_e;
649  }
650
651  double init_Dl(void)
652  {
653    /*------------------------------------- function variables -------------------------------------*/
654    double ne, Te;
655    static double Dl = 0.0;
656
657    /*---------------------------------------- function body ----------------------------------------*/
```

```
658
659   if (Dl == 0.0) {
660     read_input_file(&ne, 7);
661     read_input_file(&Te, 8);
662     Dl = sqrt(CST_EPSILON*CST_KB*Te/(ne*CST_E*CST_E));
663   }
664   return Dl;
665 }
666
667 int init_n_ini(void)
668 {
669   /*---------------------------------------- function variables ----------------------------------------*/
670   static int n_ini = -1;
671
672   /*------------------------------------------ function body --------------------------------------------*/
673
674   if (n_ini < 0) read_input_file(&n_ini, 2);
675   return n_ini;
676 }
677
678 int init_n_prev(void)
679 {
680   /*---------------------------------------- function variables ----------------------------------------*/
681   static int n_prev = -1;
682
683   /*------------------------------------------ function body --------------------------------------------*/
684
685   if (n_prev < 0) read_input_file(&n_prev, 3);
686   return n_prev;
687 }
688
689 int init_n_save(void)
690 {
691   /*---------------------------------------- function variables ----------------------------------------*/
692   static int n_save = -1;
693
694   /*------------------------------------------ function body --------------------------------------------*/
695
696   if (n_save < 0) read_input_file(&n_save, 4);
697   return n_save;
698 }
699
700 int init_n_fin(void)
701 {
702   /*---------------------------------------- function variables ----------------------------------------*/
703   static int n_fin = -1;
704
705   /*------------------------------------------ function body --------------------------------------------*/
706
707   if (n_fin < 0) read_input_file(&n_fin, 5);
708   return n_fin;
709 }
710
711 int init_n_bin_ddf(void)
712 {
713   /*---------------------------------------- function variables ----------------------------------------*/
714   static int n_bin_ddf = -1;
715
716   /*------------------------------------------ function body --------------------------------------------*/
717
718   if (n_bin_ddf < 0) read_input_file(&n_bin_ddf, 20);
719   return n_bin_ddf;
720 }
721
722 int init_n_bin_vdf(void)
723 {
724   /*---------------------------------------- function variables ----------------------------------------*/
725   static int n_bin_vdf = -1;
726
727   /*------------------------------------------ function body --------------------------------------------*/
728
729   if (n_bin_vdf < 0) read_input_file(&n_bin_vdf, 22);
730   return n_bin_vdf;
731 }
732
733 int init_n_vdf(void)
734 {
735   /*---------------------------------------- function variables ----------------------------------------*/
736   static int n_vdf = -1;
737
738   /*------------------------------------------ function body --------------------------------------------*/
739
740   if (n_vdf < 0) read_input_file(&n_vdf, 21);
741   return n_vdf;
742 }
743
744 double init_vth_e(void)
```

```
745 {
746   /*------------------------------------- function variables -------------------------------------*/
747   static double kte = init_kte();          // thermal energy of electrons
748   static double me = init_me();            // electron mass
749   static double vth_e = sqrt(2*kte/me);    // thermal velocity of electrons
750 
751   /*--------------------------------------- function body -----------------------------------------*/
752 
753   return vth_e;
754 }
755 
756 double init_vth_i(void)
757 {
758   /*------------------------------------- function variables -------------------------------------*/
759   static double kti = init_kti();          // thermal energy of ions
760   static double mi = init_mi();            // ion mass
761   static double vth_i = sqrt(2*kti/mi);    // thermal velocity of ions
762 
763   /*--------------------------------------- function body -----------------------------------------*/
764 
765   return vth_i;
766 }
767 
768 double init_v_max_e(void)
769 {
770   /*------------------------------------- function variables -------------------------------------*/
771   static double v_max_e = 0;    // max velocity to consider in velocity histograms
772 
773   /*--------------------------------------- function body -----------------------------------------*/
774 
775   if (v_max_e == 0) read_input_file(&v_max_e, 23);
776   return v_max_e;
777 }
778 
779 double init_v_min_e(void)
780 {
781   /*------------------------------------- function variables -------------------------------------*/
782   static double v_min_e = 0;    // min velocity to consider in velocity histograms
783 
784   /*--------------------------------------- function body -----------------------------------------*/
785 
786   if (v_min_e == 0) read_input_file(&v_min_e, 24);
787   return v_min_e;
788 }
789 
790 double init_v_max_i(void)
791 {
792   /*------------------------------------- function variables -------------------------------------*/
793   static double v_max_i = 0;    // max velocity to consider in velocity histograms
794 
795   /*--------------------------------------- function body -----------------------------------------*/
796 
797   if (v_max_i == 0) read_input_file(&v_max_i, 25);
798   return v_max_i;
799 }
800 
801 double init_v_min_i(void)
802 {
803   /*------------------------------------- function variables -------------------------------------*/
804   static double v_min_i = 0;    // min velocity to consider in velocity histograms
805 
806   /*--------------------------------------- function body -----------------------------------------*/
807 
808   if (v_min_i == 0) read_input_file(&v_min_i, 26);
809   return v_min_i;
810 }
811 
812 bool calibration_is_on(void)
813 {
814   /*------------------------------------- function variables -------------------------------------*/
815   static int calibration_int = -1;
816 
817   /*--------------------------------------- function body -----------------------------------------*/
818 
819   if (calibration_int < 0) {
820     read_input_file(&calibration_int, 28);
821     if (calibration_int != 0 && calibration_int != 1) {
822       cout << "Found␣error␣in␣input_data␣file.␣Wrong␣ion_current_calibration!\nStoping␣simulation.\n"
823            << endl;
824       exit(1);
825     }
826   }
827   if (calibration_int == 1) return true;
828   else return false;
829 }
830 
831 bool floating_potential_is_on(void)
```

```
832 {
833   /*------------------------------------- function variables ---------------------------------------*/
834   static int floating_potential_int = -1;
835 
836   /*---------------------------------------- function body -----------------------------------------*/
837 
838   if (floating_potential_int < 0) {
839     read_input_file(&floating_potential_int , 30);
840     if (floating_potential_int != 0 && floating_potential_int != 1) {
841       cout << "Found␣error␣in␣input_data␣file.␣Wrong␣floating_potential!\nStoping␣simulation.\n"
842            << endl;
843       exit(1);
844     }
845   }
846   if (floating_potential_int == 1) return true;
847   else return false;
848 }
849 
850 /************************************ DEVICE KERNELS DEFINITIONS ***********************************/
851 
852 __global__ void init_philox_state(curandStatePhilox4_32_10_t *state)
853 {
854   /*-------------------------- kernel variables ----------------------*/
855 
856   // kernel shared memory
857 
858   // kernel registers
859   int tid = (int) threadIdx.x + (int) blockIdx.x * (int) blockDim.x;
860   curandStatePhilox4_32_10_t local_state;
861 
862   /*-------------------------- kernel body ---------------------------*/
863 
864   // load states in local memory
865   local_state = state[tid];
866 
867   // initialize each thread state (seed, second seed, offset, pointer to state)
868   curand_init (0, tid, 0, &local_state);
869 
870   // store initialized states in global memory
871   state[tid] = local_state;
872 
873   return;
874 }
875 
876 __global__ void create_particles_kernel(particle *g_p, int num_p, double kt, double m, double L,
877                                         curandStatePhilox4_32_10_t *state)
878 {
879   /*-------------------------- kernel variables ----------------------*/
880 
881   // kernel shared memory
882 
883   // kernel registers
884   particle reg_p;
885   double sigma = sqrt(kt/m);
886   int tid = (int) threadIdx.x + (int) blockIdx.x * (int) blockDim.x;
887   int bdim = (int) blockDim.x;
888   curandStatePhilox4_32_10_t local_state;
889   double rnd;
890 
891   /*-------------------------- kernel body ---------------------------*/
892 
893   //---- load philox states from global memory
894   local_state = state[tid];
895 
896   //---- create particles
897   for (int i = tid; i < num_p; i+=bdim) {
898     rnd = curand_uniform_double(&local_state);
899     reg_p.r = rnd*L;
900     rnd = curand_normal_double(&local_state);
901     reg_p.v = rnd*sigma;
902     // store particles in global memory
903     g_p[i] = reg_p;
904   }
905   __syncthreads();
906 
907   //---- store philox states in global memory
908   state[tid] = local_state;
909 
910   return;
911 }
912 
913 __global__ void fix_velocity(double q, double m, int num_p, particle *g_p, double dt, double ds,
914                              int nn, double *g_E)
915 {
916   /*-------------------------- kernel variables ----------------------*/
917 
918   // kernel shared memory
```

```
919    double *sh_E = (double *) sh_mem;
920
921    // kernel registers
922    int tid = (int) threadIdx.x;  // thread Id
923    int bdim = (int) blockDim.x;  // block dimension
924    particle reg_p;               // register particles
925    int ic;                       // cell index
926    double dist;                  // distance from particle to nearest down vertex (normalized to ds)
927    double F;                     // force suffered for each register particle
928
929    /*--------------------------- kernel body ----------------------------*/
930
931    //---- load electric field in shared memory
932    for (int i = tid; i<nn; i+=bdim) {
933      sh_E[i] = g_E[i];
934    }
935    __syncthreads();
936
937    //---- load and analize and fix particles
938    for (int i = tid; i<num_p; i += bdim) {
939      // load particles from global to shared memory
940      reg_p = g_p[i];
941
942      // analize particles
943      ic = __double2int_rd(reg_p.r/ds);
944
945      // evaluate particle forces
946      dist = fabs(reg_p.r-ic*ds)/ds;
947      F = q*(sh_E[ic]*(1-dist)+sh_E[ic+1]*dist);
948
949      // fix particle velocities
950      reg_p.v -= 0.5*dt*F/m;
951
952      // store back particles in global memory
953      g_p[i] = reg_p;
954    }
955
956    return;
957  }
```

Code B.2: CUPIC1D1V_PP source file init.cu

```
1   /**************************************************************************************************
2    *                                                                                                *
3    *    This file is part of CUPIC1D1V_PP, a code that simulates the interaction between a plasma and  *
4    *    a planar Langmuir probe in 1D using PIC techniques accelerated with the use of GPU hardware    *
5    *    (CUDA, extension of C/C++)                                                                  *
6    *                                                                                                *
7    **************************************************************************************************/
8
9   #ifndef INIT_H
10  #define INIT_H
11
12  /********************************************* HEADERS ********************************************/
13
14  #include "stdh.h"
15  #include "random.h"
16  #include "mesh.h"
17  #include "particles.h"
18  #include "dynamic_sh_mem.h"
19  #include "cuda.h"
20
21  /************************************** SIMBOLIC CONSTANTS ****************************************/
22
23  #define CST_ME 9.109e-31      // electron mass (kg)
24  #define CST_E 1.602e-19       // electron charge (C)
25  #define CST_KB 1.381e-23      // boltzmann constant (m^2 kg s^-2 K^-1)
26  #define CST_EPSILON 8.854e-12 // free space electric permittivity (s^2 C^2 m^-3 kg^-1)
27
28  /************************************* FUNCTION PROTOTIPES ****************************************/
29
30  // host functions
31  void init_dev(void);
32  void init_sim(double **d_rho, double **d_phi, double **d_E, double **d_avg_rho, double **d_avg_phi,
33                double **d_avg_E, particle **d_e, int *num_e, particle **d_i, int *num_i,
34                double **d_avg_ddf_e, double **d_avg_vdf_e, double **d_avg_ddf_i, double **d_avg_vdf_i,
35                double *t, curandStatePhilox4_32_10_t **state);
36  void create_particles(particle **d_i, int *num_i, particle **d_e, int *num_e,
37                        curandStatePhilox4_32_10_t **state);
38  void initialize_mesh(double **d_rho, double **d_phi, double **d_E, particle *d_i, int num_i,
39                       particle *d_e, int num_e);
40  void initialize_avg_mesh(double **d_avg_rho, double **d_avg_phi, double **d_avg_E);
41  void initialize_avg_df(double **d_avg_ddf_e, double **d_avg_vdf_e, double **d_avg_ddf_i,
42                         double **d_avg_vdf_i);
43  void adjust_leap_frog(particle *d_i, int num_i, particle *d_e, int num_e, double *d_E);
44  void load_particles(particle **d_i, int *num_i, particle **d_e, int *num_e,
45                      curandStatePhilox4_32_10_t **state);
```

```
void read_particle_file(string filename, particle **d_p, int *num_p);
template <typename type> void read_input_file(type *data, int n);
double init_qi(void);
double init_qe(void);
double init_mi(void);
double init_me(void);
double init_kti(void);
double init_kte(void);
double init_vd_i(void);
double init_vd_e(void);
double init_phi_p(void);
double init_n(void);
double init_L(void);
double init_ds(void);
double init_dt(void);
double init_dtin_i(void);
double init_dtin_e(void);
double init_epsilon0(void);
int init_nc(void);
int init_nn(void);
double init_Dl(void);
int init_n_ini(void);
int init_n_prev(void);
int init_n_save(void);
int init_n_fin(void);
int init_n_bin_ddf(void);
int init_n_bin_vdf(void);
int init_n_vdf(void);
double init_vth_e(void);
double init_vth_i(void);
double init_v_max_e(void);
double init_v_min_e(void);
double init_v_max_i(void);
double init_v_min_i(void);
bool calibration_is_on(void);
bool floating_potential_is_on(void);

// device kernels
__global__ void init_philox_state(curandStatePhilox4_32_10_t *state);
__global__ void create_particles_kernel(particle *g_p, int num_p, double kt, double m, double L,
                                        curandStatePhilox4_32_10_t *state);
__global__ void fix_velocity(double q, double m, int num_p, particle *g_p, double dt, double ds,
                             int nn, double *g_E);

#endif
```

Code B.3: CUPIC1D1V_PP source file init.h

B.3. Mesh module

This is the module in charge of all the mesh related algorithms, including: particle weighting, Poisson's equation solver, and field derivation. (sources: mesh.cu, mesh.h)

```
/****************************************************************************************************
 *                                                                                                  *
 *    This file is part of CUPIC1D1V_PP, a code that simulates the interaction between a plasma and  *
 *    a planar Langmuir probe in 1D using PIC techniques accelerated with the use of GPU hardware    *
 *    (CUDA, extension of C/C++)                                                                    *
 *                                                                                                  *
 ****************************************************************************************************/

/********************************************* HEADERS *********************************************/

#include "mesh.h"

/********************************** HOST FUNCTION DEFINITIONS **********************************/

void charge_deposition(double *d_rho, particle *d_e, int num_e, particle *d_i, int num_i)
{
  /*-------------------------------------- function variables --------------------------------------*/
  // host memory
  static const double ds = init_ds();   // spatial step
  static const int nn = init_nn();      // number of nodes
  dim3 griddim, blockdim;
  size_t sh_mem_size;
  cudaError_t cuError;

  // device memory

  /*---------------------------------------- function body -----------------------------------------*/

  // initialize device memory to zeros
```

```
30    cuError = cudaMemset(d_rho, 0, nn*sizeof(double));
31    cu_check(cuError, __FILE__, __LINE__);
32
33    // set size of shared memory for particle_to_grid kernel
34    sh_mem_size = nn*sizeof(double);
35
36    // set dimensions of grid of blocks and block of threads for particle_to_grid kernel (electrons)
37    blockdim = CHARGE_DEP_BLOCK_DIM;
38    griddim = int(num_e/CHARGE_DEP_BLOCK_DIM)+1;
39
40    // call to particle_to_grid kernel (electrons)
41    cudaGetLastError();
42    particle_to_grid<<<griddim, blockdim, sh_mem_size>>>(ds, nn, d_rho, d_e, num_e, -1.0);
43    cu_sync_check(__FILE__, __LINE__);
44
45    // set dimensions of grid of blocks and block of threads for particle_to_grid kernel (ions)
46    blockdim = CHARGE_DEP_BLOCK_DIM;
47    griddim = int(num_i/CHARGE_DEP_BLOCK_DIM)+1;
48
49    // call to particle_to_grid kernel (ions)
50    cudaGetLastError();
51    particle_to_grid<<<griddim, blockdim, sh_mem_size>>>(ds, nn, d_rho, d_i, num_i, 1.0);
52    cu_sync_check(__FILE__, __LINE__);
53
54    return;
55  }
56
57  void poisson_solver(double max_error, double *d_rho, double *d_phi)
58  {
59    /*------------------------------------ function variables --------------------------------------*/
60    // host memory
61    static const double ds = init_ds();                // spatial step
62    static const int nn = init_nn();                   // number of nodes
63    static const double epsilon0 = init_epsilon0();    // electric permitivity of free space
64
65    double *h_error;
66    double t_error = max_error*10;
67    int min_iteration = 2*nn;
68
69    dim3 blockdim, griddim;
70    size_t sh_mem_size;
71    cudaError_t cuError;
72
73    // device memory
74    double *d_error;
75
76    /*--------------------------------------- function body -----------------------------------------*/
77
78    // set dimensions of grid of blocks and blocks of threads for jacobi kernel
79    blockdim = JACOBI_BLOCK_DIM;
80    griddim = (int) ((nn-2)/JACOBI_BLOCK_DIM) + 1;
81
82    // define size of shared memory for jacobi_iteration kernel
83    sh_mem_size = (2*JACOBI_BLOCK_DIM+2)*sizeof(double);
84
85    // allocate host and device memory for vector of errors
86    cuError = cudaMalloc((void **) &d_error, griddim.x*sizeof(double));
87    cu_check(cuError, __FILE__, __LINE__);
88    h_error = (double*) malloc(griddim.x*sizeof(double));
89
90    // execute jacobi iterations until solved
91    while(min_iteration>=0 || t_error>=max_error) {
92      // launch kernel for performing one jacobi iteration
93      cudaGetLastError();
94      jacobi_iteration<<<griddim, blockdim, sh_mem_size>>>(nn, ds, epsilon0, d_rho, d_phi, d_error);
95      cu_sync_check(__FILE__, __LINE__);
96
97      // copy error vector from  device to host memory
98      cuError = cudaMemcpy(h_error, d_error, griddim.x*sizeof(double), cudaMemcpyDeviceToHost);
99      cu_check(cuError, __FILE__, __LINE__);
100
101     // evaluate max error of the iteration
102     t_error = 0;
103     for (int i = 0; i<griddim.x; i++)
104     {
105       if (h_error[i] > t_error) t_error = h_error[i];
106     }
107
108     // actualize counter
109     min_iteration--;
110   }
111
112   // free device memory
113   cudaFree(d_error);
114   free(h_error);
115
116   return;
```

```
117 }
118
119 void field_solver(double *d_phi, double *d_E)
120 {
121   /*--------------------------------------- function variables ---------------------------------------*/
122   // host memory
123   static const double ds = init_ds();   // spatial step
124   static const int nn = init_nn();      // number of nodes
125   dim3 blockdim, griddim;
126
127   // device memory
128
129   /*----------------------------------------- function body -----------------------------------------*/
130
131   // set dimensions of grid of blocks and blocks of threads for jacobi kernel
132   blockdim = JACOBI_BLOCK_DIM;
133   griddim = (int) ((nn-2)/JACOBI_BLOCK_DIM) + 1;
134
135   // launch kernel for performing the derivation of the potential to obtain the electric field
136   cudaGetLastError();
137   field_derivation<<<griddim, blockdim>>>(nn, ds, d_phi, d_E);
138   cu_sync_check(__FILE__, __LINE__);
139
140   return;
141 }
142
143 /************************************ DEVICE KERNELS DEFINITIONS ***********************************/
144
145 __global__ void particle_to_grid(double ds, int nn, double *g_rho, particle *g_p, int num_p, double q)
146 {
147   /*--------------------------------------- kernel variables ----------------------------------------*/
148   // kernel shared memory
149   double *sh_partial_rho = (double *) sh_mem;   // partial rho of each bin
150
151   // kernel registers
152   int tidx = (int) threadIdx.x;
153   int tid = (int) (threadIdx.x + blockIdx.x*blockDim.x);
154   int bdim = (int) blockDim.x;
155   int ic;                           // cell index of each particle
156   particle reg_p;                   // register copy of particle analized
157   double dist;                      // distance to down vertex of the cell
158
159   /*----------------------------------------- kernel body ------------------------------------------*/
160
161   //---- initialize shared memory variables
162
163   // initialize charge density in shared memory to 0.0
164   for (int i = tidx; i < nn; i+=bdim) {
165     sh_partial_rho[i] = 0.0;
166   }
167   __syncthreads();
168
169   //--- deposition of charge
170
171   if (tid < num_p) {
172     // load particle in registers
173     reg_p = g_p[tid];
174     // calculate what cell the particle is in
175     ic = __double2int_rd(reg_p.r/ds);
176     if (reg_p.r == (nn-1)*ds) ic = nn-2;
177     if (ic >= nn-1) printf("error␣2␣on␣tid␣=␣%d,␣ic␣=␣%d,␣p.r␣=␣%f\n", tidx, ic, reg_p.r);
178     // calculate distances from particle to down vertex of the cell
179     dist = fabs(__int2double_rn(ic)*ds-reg_p.r)/ds;
180     // acumulate charge in partial rho
181     atomicAdd(&sh_partial_rho[ic], q*(1.0-dist));    //down vertex
182     atomicAdd(&sh_partial_rho[ic+1], q*dist);        //upper vertex
183   }
184   __syncthreads();
185
186   //---- volume correction (shared memory)
187
188   for (int i = tidx+1; i < nn-1; i+=bdim) {
189     sh_partial_rho[i] /= ds*ds*ds;
190   }
191   if (tidx == 0) {
192     sh_partial_rho[0] /= 0.5*ds*ds*ds;
193     sh_partial_rho[nn-1] /= 0.5*ds*ds*ds;
194   }
195   __syncthreads();
196
197   //---- charge acumulation in global memory
198
199   for (int i = tidx; i < nn; i+=bdim) {
200     atomicAdd(&g_rho[i], sh_partial_rho[i]);
201   }
202   __syncthreads();
203
```

```
204    return;
205  }
206
207  __global__ void jacobi_iteration (int nn, double ds, double epsilon0, double *g_rho, double *g_phi,
208                                    double *g_error)
209  {
210    /*--------------------------------------- kernel variables ----------------------------------------*/
211    // shared memory
212    double *sh_old_phi= (double *) sh_mem;                              //
213    double *sh_error = (double *) &sh_old_phi[JACOBI_BLOCK_DIM+2];    // manually set up shared memory
214
215    // registers
216    double new_phi, dummy_rho;
217    int tid = (int) threadIdx.x;
218    int sh_tid = (int) threadIdx.x + 1;
219    int g_tid = (int) (threadIdx.x + blockDim.x * blockIdx.x) + 1;
220    int bdim = (int) blockDim.x;
221    int bid = (int) blockIdx.x;
222    int gdim = (int) gridDim.x;
223
224    /*----------------------------------------- kernel body -------------------------------------------*/
225
226    // load phi data from global to shared memory
227    if (g_tid < nn - 1) sh_old_phi[sh_tid] = g_phi[g_tid];
228
229    // load comunication zones
230    if (bid < gdim-1) {
231      if (sh_tid == 1) sh_old_phi[sh_tid-1] = g_phi[g_tid-1];
232      if (sh_tid == bdim) sh_old_phi[sh_tid+1] = g_phi[g_tid+1];
233    } else {
234      if (sh_tid == 1) sh_old_phi[sh_tid-1] = g_phi[g_tid-1];
235      if (g_tid == nn-2) sh_old_phi[sh_tid+1] = g_phi[g_tid+1];
236    }
237    __syncthreads();
238
239    // load charge density data into registers
240    if (g_tid < nn - 1) dummy_rho = ds*ds*g_rho[g_tid]/epsilon0;
241    __syncthreads();
242
243    // actualize interior mesh points
244    if (g_tid < nn - 1) new_phi = 0.5*(dummy_rho + sh_old_phi[sh_tid-1] + sh_old_phi[sh_tid+1]);
245    __syncthreads();
246
247    // store new values of phi in global memory
248    if (g_tid < nn - 1) g_phi[g_tid] = new_phi;
249    __syncthreads();
250
251    // evaluate local errors
252    if (g_tid < nn - 1) sh_error[tid] = fabs(new_phi-sh_old_phi[sh_tid]);
253    __syncthreads();
254
255    // reduction for obtaining maximum error in current block
256    for (int stride = 1; stride < bdim; stride <<= 1) {
257      if ((tid%(stride*2) == 0) && (tid+stride < bdim) && (g_tid+stride < nn-1)) {
258        if (sh_error[tid]<sh_error[tid+stride]) sh_error[tid] = sh_error[tid+stride];
259      }
260      __syncthreads();
261    }
262
263    // store maximun error in global memory
264    if (tid == 0) g_error[bid] = sh_error[tid];
265
266    return;
267  }
268
269  __global__ void field_derivation (int nn, double ds, double *g_phi, double *g_E)
270  {
271    /*--------------------------------------- kernel variables ----------------------------------------*/
272    // shared memory
273    __shared__ double sh_phi[JACOBI_BLOCK_DIM+2];
274
275    // registers
276    double reg_E;
277    int sh_tid = (int) threadIdx.x + 1;
278    int g_tid = (int) (threadIdx.x + blockDim.x * blockIdx.x) + 1;
279    int bdim = (int) blockDim.x;
280    int bid = (int) blockIdx.x;
281    int gdim = (int) gridDim.x;
282
283    /*----------------------------------------- kernel body -------------------------------------------*/
284
285    // load phi data from global to shared memory
286    if (g_tid < nn - 1) {
287      sh_phi[sh_tid] = g_phi[g_tid];
288    }
289    // load comunication zones
290    if (bid < gdim-1) {
```

```
291      if (sh_tid == 1) sh_phi[0] = g_phi[g_tid-1];
292      if (sh_tid == bdim) sh_phi[sh_tid+1] = g_phi[g_tid+1];
293    } else {
294      if (sh_tid == 1) sh_phi[sh_tid-1] = g_phi[g_tid-1];
295      if (g_tid == nn-1) sh_phi[sh_tid] = g_phi[g_tid];
296    }
297    __syncthreads();
298
299    // calculate electric fields in interior points
300    if (g_tid < nn - 1) {
301      reg_E = (sh_phi[sh_tid-1]-sh_phi[sh_tid+1])/(2.0*ds);
302    }
303    __syncthreads();
304
305    // store electric fields of interior points in global memory
306    if (g_tid < nn - 1) g_E[g_tid] = reg_E;
307
308    // calculate electric fields at proble and plasma
309    if (g_tid == nn-1) {
310      reg_E = (sh_phi[sh_tid-1]-sh_phi[sh_tid])/ds;
311      g_E[g_tid] = reg_E;
312    } else if (g_tid == 1) {
313      reg_E = (sh_phi[sh_tid-1]-sh_phi[sh_tid])/ds;
314      g_E[g_tid-1] = reg_E;
315    }
316
317    return;
318  }
```

Code B.4: CUPIC1D1V_PP source file mesh.cu

```
1  /****************************************************************************************************
2   *                                                                                                  *
3   *    This file is part of CUPIC1D1V_PP, a code that simulates the interaction between a plasma and  *
4   *    a planar Langmuir probe in 1D using PIC techniques accelerated with the use of GPU hardware    *
5   *    (CUDA, extension of C/C++)                                                                    *
6   *                                                                                                  *
7   ****************************************************************************************************/
8
9  #ifndef MESH_H
10 #define MESH_H
11
12 /********************************************** HEADERS *********************************************/
13
14 #include "stdh.h"
15 #include "init.h"
16 #include "dynamic_sh_mem.h"
17 #include "cuda.h"
18
19 /**************************************** SIMBOLIC CONSTANTS ****************************************/
20
21 #define CHARGE_DEP_BLOCK_DIM 512   //block dimension for particle2grid kernel
22 #define JACOBI_BLOCK_DIM 128       //block dimension for jacovi_iteration kernel
23
24 /*************************************** FUNCTION PROTOTIPES ****************************************/
25
26 // host function
27 void charge_deposition(double *d_rho, particle *d_e, int num_e, particle *d_i, int num_i);
28 void poisson_solver(double max_error, double *d_rho, double *d_phi);
29 void field_solver(double *d_phi, double *d_E);
30
31 // device kernels
32 __global__ void particle_to_grid(double ds, int nn, double *g_rho, particle *g_p, int num_p, double q);
33 __global__ void jacobi_iteration (int nn, double ds, double epsilon0, double *g_rho, double *g_phi,
34                                   double *g_error);
35 __global__ void field_derivation (int nn, double ds, double *g_phi, double *g_E);
36
37 // device functions
38
39 #endif
```

Code B.5: CUPIC1D1V_PP source file mesh.h

B.4. Particles module

This is the module that manage the particles motion. It includes the field weighting algorithm as well as the particle mover, *i. e.* leap-frog algorithm. (sources: particles.cu, particles.h)

```
1  /****************************************************************************************************
2   *                                                                                                  *
3   *    This file is part of CUPIC1D1V_PP, a code that simulates the interaction between a plasma and  *
```

```
 *    a planar Langmuir probe in 1D using PIC techniques accelerated with the use of GPU hardware    *
 *    (CUDA, extension of C/C++)                                                                    *
 *                                                                                                  *
 ****************************************************************************************************/

/********************************************** HEADERS *********************************************/

#include "particles.h"

/************************************ HOST FUNCTION DEFINITIONS ************************************/

void particle_mover(particle *d_e, int num_e, particle *d_i, int num_i, double *d_E)
{
  /*------------------------------------- function variables --------------------------------------*/
  // host memory
  static const double me = init_me();     // electron's mass
  static const double mi = init_mi();     // ion's mass
  static const double qe = init_qe();     // electron's charge
  static const double qi = init_qi();     // ions's charge
  static const double ds = init_ds();     // spatial step
  static const double dt = init_dt();     // time step
  static const int nn = init_nn();        // number of nodes

  dim3 griddim, blockdim;
  size_t sh_mem_size;

  // device memory

  /*--------------------------------------- function body -----------------------------------------*/

  // set size of __shared__ memory for leap_frog kernel
  sh_mem_size = nn*sizeof(double);

  //---- move electrons

  // set dimensions of grid of blocks and blocks of threads for leap_frog kernel
  blockdim = PAR_MOV_BLOCK_DIM;
  griddim = int(num_e/PAR_MOV_BLOCK_DIM)+1;

  // call to leap_frog_step kernel (electrons)
  cudaGetLastError();
  leap_frog_step<<<griddim, blockdim, sh_mem_size>>>(qe, me, num_e, d_e, dt, ds, nn, d_E);
  cu_sync_check(__FILE__, __LINE__);

  //---- move ions

  // set dimensions of grid of blocks and blocks of threads for leap_frog kernel
  blockdim = PAR_MOV_BLOCK_DIM;
  griddim = int(num_i/PAR_MOV_BLOCK_DIM)+1;

  // call to leap_frog_step kernel (ions)
  cudaGetLastError();
  leap_frog_step<<<griddim, blockdim, sh_mem_size>>>(qi, mi, num_i, d_i, dt, ds, nn, d_E);
  cu_sync_check(__FILE__, __LINE__);

  return;
}

/************************************ DEVICE KERNELS DEFINITIONS ***********************************/

__global__ void leap_frog_step(double q, double m, int num_p, particle *g_p, double dt, double ds,
                               int nn, double *g_E)
{
  /*-------------------------------------- kernel variables ---------------------------------------*/
  // kernel shared memory
  double *sh_E = (double *) sh_mem;   // manually set up shared memory variables

  // kernel registers
  int tidx = (int) threadIdx.x;
  int tid = (int) threadIdx.x + (int) blockDim.x * (int) blockIdx.x;  // thread Id
  int bdim = (int) blockDim.x;  // block dimension
  particle reg_p;               // register particles
  int ic;                       // cell index
  double dist;                  // distance from particle to nearest down vertex (normalized to ds)
  double F;                     // force suffered for each register particle

  /*----------------------------------------- kernel body -----------------------------------------*/

  //---- initialize shared memory variables

  // load fields from global memory
  for (int i = tidx; i<nn; i += bdim) {
    sh_E[i] = g_E[i];
  }
  __syncthreads();

  //---- Process batches of particles
```

```
  if (tid < num_p) {
    // load particle data in registers
    reg_p = g_p[tid];

    // find cell index
    ic = __double2int_rd(reg_p.r/ds);

    // evaluate distance to nearest down vertex (normalized to ds)
    dist = fabs(reg_p.r-ic*ds)/ds;

    // calculate particle's forces
    F = q*(sh_E[ic]*(1.0-dist) + sh_E[ic+1]*dist);

    // move particles
    reg_p.v += dt*F/m;
    reg_p.r += dt*reg_p.v;

    // store particle data in global memory
    g_p[tid] = reg_p;
  }

  return;
}
```

Code B.6: CUPIC1D1V_PP source file particles.cu

```
/****************************************************************************************************
 *                                                                                                  *
 *    This file is part of CUPIC1D1V_PP, a code that simulates the interaction between a plasma and  *
 *    a planar Langmuir probe in 1D using PIC techniques accelerated with the use of GPU hardware    *
 *    (CUDA, extension of C/C++)                                                                    *
 *                                                                                                  *
 ****************************************************************************************************/

#ifndef PARTICLES_H
#define PARTICLES_H

/********************************************** HEADERS **********************************************/

#include "stdh.h"
#include "init.h"
#include "diagnostic.h"
#include "dynamic_sh_mem.h"
#include "cuda.h"

/**************************************** SIMBOLIC CONSTANTS ****************************************/

#define PAR_MOV_BLOCK_DIM 512   //block dimension for defragmentation kernel

/*************************************** FUNCTION PROTOTIPES ****************************************/

// host function
void particle_mover(particle *d_e, int num_e, particle *d_i, int num_i, double *d_E);

// device kernels
__global__ void leap_frog_step(double q, double m, int num_p, particle *g_p, double dt, double ds,
                               int nn, double *g_E);

// device functions

#endif
```

Code B.7: CUPIC1D1V_PP source file particles.h

B.5. Boundary conditions module

This is the module that takes care for the influx of particles coming from the plasma, as well as the absorption of particles at both boundaries. (sources: cc.cu, cc.h)

```
/****************************************************************************************************
 *                                                                                                  *
 *    This file is part of CUPIC1D1V_PP, a code that simulates the interaction between a plasma and  *
 *    a planar Langmuir probe in 1D using PIC techniques accelerated with the use of GPU hardware    *
 *    (CUDA, extension of C/C++)                                                                    *
 *                                                                                                  *
 ****************************************************************************************************/

/********************************************** HEADERS **********************************************/

#include "cc.h"
```

```
12
13 /************************************* HOST FUNCTION DEFINITIONS *************************************/
14
15 void cc (double t, int *num_e, particle **d_e, double *dtin_e, double *vd_e, int *num_i,
16          particle **d_i, double *dtin_i, double *vd_i, double *q_p, double *d_phi, double *d_E,
17          curandStatePhilox4_32_10_t *state)
18 {
19   /*------------------------------------- function variables --------------------------------------*/
20
21   // host memory
22   static const double me = init_me();                  //
23   static const double mi = init_mi();                  // particle
24   static const double kte = init_kte();                // properties
25   static const double kti = init_kti();                //
26
27   static double tin_e = t+(*dtin_e);                   // time for next electron insertion
28   static double tin_i = t+(*dtin_i);                   // time for next ion insertion
29
30   static bool fp_is_on = floating_potential_is_on();  // probe is floating or not
31   static bool flux_cal_is_on = calibration_is_on();   // probe is floating or not
32   static int nc = init_nc();                           // number of cells
33   static double ds = init_ds();                        // spatial step
34   static double epsilon0 = init_epsilon0();            // epsilon0 in simulation units
35
36   double phi_s;                                        // sheath edge potential
37   double phi_p;                                        // probe potential
38
39   cudaError cuError;                                   // cuda error variable
40
41   // device memory
42
43   /*--------------------------------------- function body -----------------------------------------*/
44
45   //---- electrons contour conditions
46   abs_emi_cc(t, &tin_e, *dtin_e, kte, *vd_e, me, -1.0, q_p,  num_e, d_e, d_E, state);
47
48   //---- ions contour conditions
49   abs_emi_cc(t, &tin_i, *dtin_i, kti, *vd_i, mi, +1.0, q_p, num_i, d_i, d_E, state);
50
51   //---- evaluate probe and sheath edge potentials in case fp or flux_cal are on
52   if (fp_is_on || flux_cal_is_on) {
53     cuError = cudaMemcpy (&phi_p, &d_phi[0], sizeof(double), cudaMemcpyDeviceToHost);
54     cu_check(cuError, __FILE__, __LINE__);
55     cuError = cudaMemcpy (&phi_s, &d_phi[nc], sizeof(double), cudaMemcpyDeviceToHost);
56     cu_check(cuError, __FILE__, __LINE__);
57   }
58
59   //---- actulize ion drift velocity in order to ensure zero field at sheath edge
60   if (flux_cal_is_on) {
61     calibrate_ion_flux(vd_i, d_E, &phi_s);
62   }
63
64   //---- actualize probe potential because of the change in probe charge
65   if (fp_is_on) {
66     phi_p = 0.5*(*q_p)*nc/(ds*epsilon0);
67     if (phi_p > phi_s) phi_p = phi_s;
68   }
69
70   //---- store new probe and sheath edge potentials in d_phi and recalculate electron and ion dtin
71   if (fp_is_on || flux_cal_is_on) {
72     cuError = cudaMemcpy (&d_phi[0], &phi_p, sizeof(double), cudaMemcpyHostToDevice);
73     cu_check(cuError, __FILE__, __LINE__);
74     cuError = cudaMemcpy (&d_phi[nc], &phi_s, sizeof(double), cudaMemcpyHostToDevice);
75     cu_check(cuError, __FILE__, __LINE__);
76     recalculate_dtin(dtin_e, dtin_i, *vd_e, *vd_i, phi_p, phi_s);
77   }
78
79   return;
80 }
81
82 void abs_emi_cc(double t, double *tin, double dtin, double kt, double vd, double m, double q,
83                 double *q_p, int *h_num_p, particle **d_p, double *d_E,
84                 curandStatePhilox4_32_10_t *state)
85 {
86   /*------------------------------------- function variables --------------------------------------*/
87
88   // host memory
89   static const double L = init_L();       //
90   static const double ds = init_ds();     // geometric properties
91   static const int nn = init_nn();        // of simulation
92
93   static const double dt = init_dt();     //
94   double fpt = t+dt;                      // timing variables
95   double fvt = t+0.5*dt;                  //
96
97   int in = 0;                             // number of particles added at plasma frontier
98   int h_num_abs_p;                        // host number of particles absorbed at the probe
```

```
99
100   cudaError cuError;                       // cuda error variable
101   dim3 griddim, blockdim;                  // kernel execution configurations
102
103   // device memory
104   int *d_num_p;                            // device number of particles
105   int *d_num_abs_p;                        // device number of particles absorved at the probe
106   particle *d_dummy_p;                     // device dummy vector for particle storage
107
108   /*--------------------------------------- function body -----------------------------------------*/
109
110   // calculate number of particles that flow into the simulation
111   if((*tin) < fpt) in = 1 + int((fpt-(*tin))/dtin);
112
113   // copy number of particles from host to device
114   cuError = cudaMalloc((void **) &d_num_p, sizeof(int));
115   cu_check(cuError, __FILE__, __LINE__);
116   cuError = cudaMemcpy (d_num_p, h_num_p, sizeof(int), cudaMemcpyHostToDevice);
117   cu_check(cuError, __FILE__, __LINE__);
118
119   // initialize number of particles absorbed at the probe
120   cuError = cudaMalloc((void **) &d_num_abs_p, sizeof(int));
121   cu_check(cuError, __FILE__, __LINE__);
122   cuError = cudaMemset((void *) d_num_abs_p, 0, sizeof(int));
123   cu_check(cuError, __FILE__, __LINE__);
124
125   // execution configuration for particle remover kernel
126   griddim = 1;
127   blockdim = P_RMV_BLK_SZ;
128
129   // execute particle remover kernel
130   cudaGetLastError();
131   pRemover<<<griddim, blockdim>>>(*d_p, d_num_p, L, d_num_abs_p);
132   cu_sync_check(__FILE__, __LINE__);
133
134   // copy number of particles absorbed at the probe from device to host (and free device memory)
135   cuError = cudaMemcpy (&h_num_abs_p, d_num_abs_p, sizeof(int), cudaMemcpyDeviceToHost);
136   cu_check(cuError, __FILE__, __LINE__);
137   cuError = cudaFree(d_num_abs_p);
138   cu_check(cuError, __FILE__, __LINE__);
139
140   // actualize probe acumulated charge
141   *q_p += q*h_num_abs_p;
142
143   // copy new number of particles from device to host (and free device memory)
144   cuError = cudaMemcpy (h_num_p, d_num_p, sizeof(int), cudaMemcpyDeviceToHost);
145   cu_check(cuError, __FILE__, __LINE__);
146   cuError = cudaFree(d_num_p);
147   cu_check(cuError, __FILE__, __LINE__);
148
149   // resize of particle vector with new number of particles
150   cuError = cudaMalloc((void **) &d_dummy_p, ((*h_num_p)+in)*sizeof(particle));
151   cu_check(cuError, __FILE__, __LINE__);
152   cuError = cudaMemcpy(d_dummy_p, *d_p, (*h_num_p)*sizeof(particle), cudaMemcpyDeviceToDevice);
153   cu_check(cuError, __FILE__, __LINE__);
154   cuError = cudaFree(*d_p);
155   cu_check(cuError, __FILE__, __LINE__);
156   cuError = cudaMalloc((void **) d_p, ((*h_num_p)+in)*sizeof(particle));
157   cu_check(cuError, __FILE__, __LINE__);
158   cuError = cudaMemcpy(*d_p, d_dummy_p, (*h_num_p)*sizeof(particle), cudaMemcpyDeviceToDevice);
159   cu_check(cuError, __FILE__, __LINE__);
160   cuError = cudaFree(d_dummy_p);
161   cu_check(cuError, __FILE__, __LINE__);
162
163   // add particles
164   if (in != 0) {
165     // execution configuration for pEmi kernel
166     griddim = 1;
167     blockdim = CURAND_BLOCK_DIM;
168
169     // launch kernel to add particles
170     cudaGetLastError();
171     pEmi<<<griddim, blockdim>>>(*d_p, *h_num_p, in, d_E, sqrt(kt/m), vd, q/m, nn, L, fpt, fvt, *tin,
172                                 dtin, state);
173     cu_sync_check(__FILE__, __LINE__);
174
175     // actualize time for next particle insertion
176     (*tin) += double(in)*dtin;
177
178     // actualize number of particles
179     *h_num_p += in;
180   }
181
182   return;
183 }
184
185 void recalculate_dtin(double *dtin_e, double *dtin_i, double vd_e, double vd_i, double phi_p,
```

```
186       double phi_s)
187 {
188   /*-------------------------------------- function variables ---------------------------------------*/
189
190   // host memory
191   static const double n = init_n();
192   static const double ds = init_ds();
193   static const double me = init_me();
194   static const double kte = init_kte();
195   static const double mi = init_mi();
196   static const double kti = init_kti();
197
198   // device memory
199
200   /*---------------------------------------- function body -----------------------------------------*/
201
202   //---- recalculate electron dtin
203   *dtin_e = n*sqrt(kte/(2.0*PI*me))*exp(-0.5*me*vd_e*vd_e/kte);  // thermal component of input flux
204   *dtin_e -= 0.5*n*vd_e*(1.0+erf(sqrt(0.5*me/kte)*(-vd_e)));     // drift component of input flux
205   *dtin_e *= exp(phi_s)*0.5*(1.0+erf(sqrt(phi_s-phi_p)));        // corrected density at sheath edge
206   *dtin_e *= ds*ds;          // number of particles that enter the simulation per unit of time
207   *dtin_e = 1.0/(*dtin_e);  // time between consecutive particles injection
208
209   //---- recalculate ion dtin
210   *dtin_i = n*sqrt(kti/(2.0*PI*mi))*exp(-0.5*mi*vd_i*vd_i/kti);  // thermal component of input flux
211   *dtin_i -= 0.5*n*vd_i*(1.0+erf(sqrt(0.5*mi/kti)*(-vd_i)));     // drift component of input flux
212   *dtin_i *= exp(phi_s)*0.5*(1.0+erf(sqrt(phi_s-phi_p)));        // corrected density at sheath edge
213   *dtin_i *= ds*ds;          // number of particles that enter the simulation per unit of time
214   *dtin_i = 1.0/(*dtin_i);  // time between consecutive particles injection
215
216   return;
217 }
218
219 void calibrate_ion_flux(double *vd_i, double *d_E, double *phi_s)
220 {
221   /*-------------------------------------- function variables ---------------------------------------*/
222
223   // host memory
224   static const double mi = init_mi();
225   static const int nc = init_nc();
226
227   double E_ps = -1.0e-2;
228   double h_Es;
229   const double increment = 1.0e-6;
230
231   cudaError cuError;                              // cuda error variable
232
233   // device memory
234
235   /*---------------------------------------- function body -----------------------------------------*/
236
237   //---- Actualize ion drift velocity acording to the value of electric field at plasma frontier
238
239   // copy field from device to host memory
240   cuError = cudaMemcpy (&h_Es, &d_E[nn-1], sizeof(double), cudaMemcpyDeviceToHost);
241   cu_check(cuError, __FILE__, __LINE__);
242
243   // actualize ion drift velocity
244   if (h_Es<E_ps && *vd_i > -2.0/sqrt(mi)) {
245     *vd_i -= increment;
246   } else if (h_Es>E_ps && *vd_i < 0.0) {
247     *vd_i += increment;
248   }
249
250   // actualize sheath edge potential
251   *phi_s = -0.5*mi*(*vd_i)*(*vd_i);
252
253   return;
254 }
255
256 /************************************ DEVICE KERNELS DEFINITIONS ************************************/
257
258 __global__ void pEmi(particle *g_p, int num_p, int n_in, double *g_E, double vth, double vd, double qm,
259                      int nn, double L, double fpt, double fvt, double tin, double dtin,
260                      curandStatePhilox4_32_10_t *state)
261 {
262   /*--------------------------------------- kernel variables ---------------------------------------*/
263
264   // kernel shared memory
265   __shared__ double sh_E;
266
267   // kernel registers
268   particle reg_p;
269   int tid = (int) threadIdx.x + (int) blockIdx.x * (int) blockDim.x;
270   int tpb = (int) blockDim.x;
271   curandStatePhilox4_32_10_t local_state;
272   double2 rnd;
```

```
273
274   /*---------------------------------------- kernel body ------------------------------------------*/
275
276   //---- initialize shared memory
277   if (tid == 0) sh_E = g_E[nn-1];
278   __syncthreads();
279
280   //---- initialize registers
281   local_state = state[tid];
282   __syncthreads();
283
284   //---- generate particles
285   for (int i = tid; i < n_in; i+=tpb) {
286     // generate register particles
287     reg_p.r = L;
288     if (vth > 0.0) {
289       rnd = curand_normal2_double(&local_state);
290       reg_p.v = -sqrt(rnd.x*rnd.x+rnd.y*rnd.y)*vth+vd;
291     } else reg_p.v = vd;
292
293     // simple push
294     reg_p.r += (fpt-(tin+double(i)*dtin))*reg_p.v;
295     reg_p.v += (fvt-(tin+double(i)*dtin))*sh_E*qm;
296
297     // store new particles in global memory
298     g_p[num_p+i] = reg_p;
299   }
300   __syncthreads();
301
302   //---- store local state in global memory
303   state[tid] = local_state;
304
305   return;
306 }
307
308 __global__ void pRemover (particle *g_p, int *g_num_p, double L, int *g_num_abs_p)
309 {
310   /*-------------------------------------- kernel variables ----------------------------------------*/
311
312   // kernel shared memory
313   __shared__ int sh_tail;
314   __shared__ int sh_num_abs_p;
315
316   // kernel registers
317   int tid = (int) threadIdx.x;
318   int bdim = (int) blockDim.x;
319   int N = *g_num_p;
320   int ite = (N/bdim)*bdim;
321   int reg_tail;
322   particle reg_p;
323
324   /*---------------------------------------- kernel body ------------------------------------------*/
325
326   //---- initialize shared memory
327   if (tid == 0) {
328     sh_tail = 0;
329     sh_num_abs_p = 0;
330   }
331   __syncthreads();
332
333   //---- analize full batches of particles
334   for (int i = tid; i<ite; i+=bdim) {
335     // load particles from global memory to registers
336     reg_p = g_p[i];
337
338     // analize particle
339     if (reg_p.r >= 0 && reg_p.r <= L) {
340       reg_tail = atomicAdd(&sh_tail, 1);
341     } else {
342       reg_tail = -1;
343       if (reg_p.r < 0.0) atomicAdd(&sh_num_abs_p, 1);
344     }
345     __syncthreads();
346
347     // store accepted particles in global memory
348     if (reg_tail >= 0) g_p[reg_tail] = reg_p;
349     __syncthreads();
350   }
351   __syncthreads();
352
353   //---- analize last batch of particles
354   if (ite+tid < N) {
355     // loag particles from global memory to registers
356     reg_p = g_p[ite+tid];
357
358     // analize particle
359     if (reg_p.r >= 0 && reg_p.r <= L) {
```

```
360        reg_tail = atomicAdd(&sh_tail, 1);
361      } else {
362        reg_tail = -1;
363        if (reg_p.r < 0.0) atomicAdd(&sh_num_abs_p, 1);
364      }
365    }
366    __syncthreads();
367
368    // store accepted particles of last batch in global memory
369    if (ite+tid < N && reg_tail >= 0) g_p[reg_tail] = reg_p;
370
371    // store new number of particles in global memory
372    if (tid == 0) {
373      *g_num_p = sh_tail;
374      *g_num_abs_p = sh_num_abs_p;
375    }
376
377    return;
378  }
```

Code B.8: CUPIC1D1V_PP source file cc.cu

```
1   /****************************************************************************************************
2    *                                                                                                  *
3    *    This file is part of CUPIC1D1V_PP, a code that simulates the interaction between a plasma and  *
4    *    a planar Langmuir probe in 1D using PIC techniques accelerated with the use of GPU hardware    *
5    *    (CUDA, extension of C/C++)                                                                    *
6    *                                                                                                  *
7    ****************************************************************************************************/
8
9   #ifndef CC_H
10  #define CC_H
11
12  /********************************************** HEADERS *********************************************/
13
14  #include "stdh.h"
15  #include "init.h"
16  #include "random.h"
17  #include "diagnostic.h"
18  #include "cuda.h"
19  #include "dynamic_sh_mem.h"
20
21  /*************************************** SIMBOLIC CONSTANTS ****************************************/
22
23  #define P_RMV_BLK_SZ 1024      //block dimension for particle remover kernel
24
25  /************************************** FUNCTION PROTOTIPES ****************************************/
26
27  // host function
28  void cc (double t, int *num_e, particle **d_e, double *dtin_e, double *vd_e, int *num_i,
29           particle **d_i, double *dtin_i, double *vd_i, double *q_p, double *d_phi, double *d_E,
30           curandStatePhilox4_32_10_t *state);
31  void abs_emi_cc(double t, double *tin, double dtin, double kt, double vd, double m, double q,
32                  double *q_p, int *h_num_p, particle **d_p, double *d_E,
33                  curandStatePhilox4_32_10_t *state);
34  void recalculate_dtin(double *dtin_e, double *dtin_i, double vd_e, double vd_i, double phi_p,
35                        double phi_s);
36  void calibrate_ion_flux(double *vd_i, double *d_E, double *phi_s);
37
38  // device kernels
39  __global__ void pEmi(particle *g_p, int num_p, int n_in, double *g_E, double vth, double vd, double qm,
40                       int nn, double L, double fpt, double fvt, double tin, double dtin,
41                       curandStatePhilox4_32_10_t *state);
42  __global__ void pRemover (particle *g_p, int *g_num_p, double L, int *g_num_abs_p);
43
44  #endif
```

Code B.9: CUPIC1D1V_PP source file cc.h

B.6. Diagnostic

This is the module that contains all the functions and algorithms that analyse raw data from the simulation on the fly, it also saves data into files for its subsequent analysis. (sources: diagnostic.cu, diagnostic.h)

```
1   /****************************************************************************************************
2    *                                                                                                  *
3    *    This file is part of CUPIC1D1V_PP, a code that simulates the interaction between a plasma and  *
4    *    a planar Langmuir probe in 1D using PIC techniques accelerated with the use of GPU hardware    *
5    *    (CUDA, extension of C/C++)                                                                    *
```

```
 *                                                                                              *
 ***********************************************************************************************/

/********************************************* HEADERS ********************************************/

#include "diagnostic.h"

/********************************** HOST FUNCTION DEFINITIONS ***********************************/

void avg_mesh(double *d_foo, double *d_avg_foo, int *count)
{
  /*------------------------------------ function variables --------------------------------------*/

  // host memory
  static const int nn = init_nn();          // number of nodes
  static const int n_save = init_n_save();  // number of iterations to average

  dim3 griddim, blockdim;
  cudaError_t cuError;

  // device memory

  /*-------------------------------------- function body ----------------------------------------*/

  // check if restart of avg_foo is needed
  if (*count == n_save) {
    //reset count
    *count = 0;

    //reset avg_foo
    cuError = cudaMemset ((void *) d_avg_foo, 0, nn*sizeof(double));
    cu_check(cuError, __FILE__, __LINE__);
  }

  // set dimensions of grid of blocks and block of threads for kernels
  blockdim = AVG_MESH_BLOCK_DIM;
  griddim = int(nn/AVG_MESH_BLOCK_DIM)+1;

  // call to mesh_sum kernel
  cudaGetLastError();
  mesh_sum<<<griddim, blockdim>>>(d_foo, d_avg_foo, nn);
  cu_sync_check(__FILE__, __LINE__);

  // actualize count
  *count += 1;

  // normalize average if reached desired number of iterations
  if (*count == n_save ) {
    cudaGetLastError();
    mesh_norm<<<griddim, blockdim>>>(d_avg_foo, (double) n_save, nn);
    cu_sync_check(__FILE__, __LINE__);
  }

  return;
}

void eval_df(double *d_avg_ddf, double *d_avg_vdf, double vmax, double vmin, particle *d_p, int num_p,
             int *count)
{
  /*------------------------------------ function variables --------------------------------------*/

  // host memory
  static const int n_bin_ddf = init_n_bin_ddf();    // number of bins density distribution functions
  static const int n_bin_vdf = init_n_bin_vdf();    // number of bins velocity distribution functions
  static const int n_vdf = init_n_vdf();            // number of velocity distribution functions
  static const int n_save = init_n_save();          // number of iterations to average
  static const double L = init_L();                 // lenght of simulation

  dim3 griddim, blockdim;
  size_t sh_mem_size;
  cudaError_t cuError;

  // device memory

  /*-------------------------------------- function body ----------------------------------------*/

  // check if restart of distribution functions is needed
  if (*count == n_save) {
    //reset count
    *count = 0;

    // reset averaged distribution functions
    cuError = cudaMemset ((void *) d_avg_ddf, 0, n_bin_ddf*sizeof(double));
    cu_check(cuError, __FILE__, __LINE__);
    cuError = cudaMemset ((void *) d_avg_vdf, 0, n_bin_vdf*n_vdf*sizeof(double));
    cu_check(cuError, __FILE__, __LINE__);
  }
```

```
93
94    // set dimensions of grid of blocks and block of threads for kernel and shared memory size
95    blockdim = PARTICLE2DF_BLOCK_DIM;
96    griddim = int(num_p/PARTICLE2DF_BLOCK_DIM) + 1;
97    sh_mem_size = sizeof(int)*(n_bin_ddf+(n_bin_vdf+1)*n_vdf);
98
99    // call to mesh_sum kernel
100   cudaGetLastError();
101   particle2df<<<griddim, blockdim, sh_mem_size>>>(d_avg_ddf, n_bin_ddf, L, d_avg_vdf, n_vdf,
102                                                  n_bin_vdf, vmax, vmin, d_p, num_p);
103   cu_sync_check(__FILE__, __LINE__);
104
105   // actualize count
106   *count += 1;
107
108   // normalize average if reached desired number of iterations
109   //if (*count == n_save ) {
110     //cudaGetLastError();
111     //kernel<<<griddim, blockdim>>>();
112     //cu_sync_check(__file__, __line__);
113   //}
114
115   return;
116 }
117
118 double eval_particle_energy(double *d_phi,  particle *d_p, double m, double q, int num_p)
119 {
120   /*---------------------------------------- function variables ----------------------------------------*/
121
122   // host memory
123   static const int nn = init_nn();        // number of nodes
124   static const double ds = init_ds();     // spacial step
125   double *h_partial_U;                    // partial energy of each block
126   double h_U = 0.0;                       // total energy of particle system
127
128   dim3 griddim, blockdim;
129   size_t sh_mem_size;
130   cudaError_t cuError;
131
132   // device memory
133   double *d_partial_U;
134
135   /*------------------------------------------ function body -------------------------------------------*/
136
137   // set execution configuration of the kernel that evaluates energy
138   blockdim = ENERGY_BLOCK_DIM;
139   griddim = int(num_p/ENERGY_BLOCK_DIM)+1;
140
141   // allocate host and device memory for block's energy
142   cuError = cudaMalloc ((void **) &d_partial_U, griddim.x*sizeof(double));
143   cu_check(cuError, __FILE__, __LINE__);
144   h_partial_U = (double *) malloc(griddim.x*sizeof(double));
145
146   // define size of shared memory for energy_kernel
147   sh_mem_size = (ENERGY_BLOCK_DIM+nn)*sizeof(double);
148
149   // launch kernel to evaluate energy of the whole system
150   cudaGetLastError();
151   energy_kernel<<<griddim, blockdim, sh_mem_size>>>(d_partial_U, d_phi, nn, ds, d_p, m, q, num_p);
152   cu_sync_check(__FILE__, __LINE__);
153
154   // copy sistem energy from device to host
155   cuError = cudaMemcpy (h_partial_U, d_partial_U, griddim.x*sizeof(double), cudaMemcpyDeviceToHost);
156   cu_check(cuError, __FILE__, __LINE__);
157
158   // reduction of block's energy
159   for (int i = 0; i<griddim.x; i++) h_U += h_partial_U[i];
160
161   //free host and device memory for block's energy
162   cuError = cudaFree(d_partial_U);
163   cu_check(cuError, __FILE__, __LINE__);
164   free(h_partial_U);
165
166   return h_U;
167 }
168
169 void particles_snapshot(particle *d_p, int num_p, string filename)
170 {
171   /*---------------------------------------- function variables ----------------------------------------*/
172
173   // host memory
174   particle *h_p;
175   FILE *pFile;
176   cudaError_t cuError;
177
178   // device memory
179
```

```
180
181   /*---------------------------------------- function body -----------------------------------------*/
182
183   // allocate host memory for particle vector
184   h_p = (particle *) malloc(num_p*sizeof(particle));
185
186   // copy particle vector from device to host
187   cuError = cudaMemcpy (h_p, d_p, num_p*sizeof(particle), cudaMemcpyDeviceToHost);
188   cu_check(cuError, __FILE__, __LINE__);
189
190   // save snapshot to file
191   filename.append(".dat");
192   pFile = fopen(filename.c_str(), "w");
193   for (int i = 0; i < num_p; i++) {
194     fprintf(pFile, "␣%.17e␣%.17e␣\n", h_p[i].r, h_p[i].v);
195   }
196   fclose(pFile);
197
198   // free host memory
199   free(h_p);
200
201   return;
202 }
203
204 void save_mesh(double *d_m, string filename)
205 {
206   /*-------------------------------------- function variables ---------------------------------------*/
207
208   // host memory
209   static const int nn = init_nn();
210   double *h_m;
211   FILE *pFile;
212   cudaError_t cuError;
213
214   // device memory
215
216
217   /*---------------------------------------- function body -----------------------------------------*/
218
219   // allocate host memory for mesh vector
220   h_m = (double *) malloc(nn*sizeof(double));
221
222   // copy particle vector from device to host
223   cuError = cudaMemcpy (h_m, d_m, nn*sizeof(double), cudaMemcpyDeviceToHost);
224   cu_check(cuError, __FILE__, __LINE__);
225
226   // save snapshot to file
227   filename.append(".dat");
228   pFile = fopen(filename.c_str(), "w");
229   for (int i = 0; i < nn; i++) {
230     fprintf(pFile, "␣%d␣%.17e␣\n", i, h_m[i]);
231   }
232   fclose(pFile);
233
234   // free host memory
235   free(h_m);
236
237   return;
238 }
239
240 void save_ddf(double *d_avg_ddf, string filename)
241 {
242   /*-------------------------------------- function variables ---------------------------------------*/
243
244   // host memory
245   static const double L = init_L();                     // size of simulation
246   static const int n_bin_ddf = init_n_bin_ddf();        // number of bins of ddf
247   static const double bin_size = L/double(n_bin_ddf);   // size of each bin
248
249   double *h_avg_ddf;                                    // host memory for ddf
250
251   FILE *pFile;
252   cudaError_t cuError;
253
254   // device memory
255
256   /*---------------------------------------- function body -----------------------------------------*/
257
258   // allocate host memory for ddf
259   h_avg_ddf = (double *) malloc(n_bin_ddf*sizeof(double));
260
261   // copy ddf from device to host
262   cuError = cudaMemcpy (h_avg_ddf, d_avg_ddf, n_bin_ddf*sizeof(double), cudaMemcpyDeviceToHost);
263   cu_check(cuError, __FILE__, __LINE__);
264
265   // save bins to file
266   filename.append(".dat");
```

```
267    pFile = fopen(filename.c_str(), "w");
268    for (int i = 0; i < n_bin_ddf; i++) {
269      fprintf(pFile, "␣%lf␣%lf␣\n", (double(i)+0.5)*bin_size, h_avg_ddf[i]);
270    }
271    fclose(pFile);
272
273    //free host memory for particle vector
274    free(h_avg_ddf);
275
276    return;
277  }
278
279  void save_vdf(double *d_avg_vdf, double vmax, double vmin, string filename)
280  {
281    /*------------------------------------- function variables ---------------------------------------*/
282
283    // host memory
284    static const double L = init_L();                      // size of simulation
285    static const int n_vdf = init_n_vdf();                 // number of vdfs
286    static const int n_bin_vdf = init_n_bin_vdf();         // number of bins of vdf
287    static const double r_bin_size = L/double(n_vdf);      // size of spatial bins
288    const double v_bin_size = (vmax-vmin)/n_bin_vdf;       // size of velocity bins
289
290    double *h_avg_vdf;                                     // host memory for ddf
291
292    FILE *pFile;
293    cudaError_t cuError;
294
295    // device memory
296
297    /*--------------------------------------- function body -----------------------------------------*/
298
299    // allocate host memory for vdf
300    h_avg_vdf = (double *) malloc(n_vdf*n_bin_vdf*sizeof(double));
301
302    // copy vdf from device to host
303    cuError = cudaMemcpy (h_avg_vdf, d_avg_vdf, n_vdf*n_bin_vdf*sizeof(double), cudaMemcpyDeviceToHost);
304    cu_check(cuError, __FILE__, __LINE__);
305
306    // save bins to file
307    filename.append(".dat");
308    pFile = fopen(filename.c_str(), "w");
309    for (int i = 0; i < n_vdf; i++) {
310      for (int j = 0; j < n_bin_vdf; j++) {
311        fprintf(pFile, "␣%g␣%g␣%g␣\n", (double(i)+0.5)*r_bin_size, (double(j)+0.5)*v_bin_size+vmin,
312                                       h_avg_vdf[j+n_bin_vdf*i]);
313      }
314      fprintf(pFile, "\n");
315    }
316    fclose(pFile);
317
318    //free host memory for particle vector
319    free(h_avg_vdf);
320
321    return;
322  }
323
324  void save_log(double t, int num_e, int num_i, double U_e, double U_i, double vd_e, double vd_i,
325                double *d_phi)
326  {
327    /*------------------------------------- function variables ---------------------------------------*/
328
329    // host memory
330    double dummy_phi_p;
331    string filename = "../output/log.dat";
332    FILE *pFile;
333
334    cudaError cuError;                        // cuda error variable
335
336    // device memory
337
338    /*--------------------------------------- function body -----------------------------------------*/
339
340    // copy probe's potential from device to host memory
341    cuError = cudaMemcpy (&dummy_phi_p, &d_phi[0], sizeof(double), cudaMemcpyDeviceToHost);
342    cu_check(cuError, __FILE__, __LINE__);
343
344    // save log to file
345    pFile = fopen(filename.c_str(), "a");
346    if (pFile == NULL) {
347      printf ("Error␣opening␣log␣file␣\n");
348      exit(1);
349    } else fprintf(pFile, "␣%.17e␣%d␣%d␣%.17e␣%.17e␣%.17e␣%.17e␣%.17e␣\n", t, num_e, num_i, U_e, U_i,
350                   vd_e, vd_i, dummy_phi_p);
351    fclose(pFile);
352
353    return;
```

```
}

double calculate_vd_i(double dtin_i)
{
  /*-------------------------------------- function variables ---------------------------------------*/

  // host memory
  static const double n = init_n();              // plasma density
  static const double ds = init_ds();            // spatial step

  // device memory

  /*--------------------------------------- function body -----------------------------------------*/

  return 1.0/(n*dtin_i*ds*ds);
}

/************************************ DEVICE KERNELS DEFINITIONS ***********************************/

__global__ void mesh_sum(double *g_foo, double *g_avg_foo, int nn)
{
  /*--------------------------------------- kernel variables ---------------------------------------*/

  // kernel shared memory

  // kernel registers
  double reg_foo, reg_avg_foo;

  int tid = (int) (threadIdx.x + blockIdx.x * blockDim.x);

  /*---------------------------------------- kernel body -------------------------------------------*/

  // load data from global memory to registers
  if (tid < nn) {
    reg_foo = g_foo[tid];
    reg_avg_foo = g_avg_foo[tid];
  }
  __syncthreads();

  // add foo to avg foo
  if (tid < nn) {
    reg_avg_foo += reg_foo;
  }
  __syncthreads();

  // store data y global memory
  if (tid < nn) {
    g_avg_foo[tid] = reg_avg_foo ;
  }

  return;
}

__global__ void mesh_norm(double *g_avg_foo, double norm_cst, int nn)
{
  /*--------------------------------------- kernel variables ---------------------------------------*/

  // kernel shared memory

  // kernel registers
  double reg_avg_foo;

  int tid = (int) (threadIdx.x + blockIdx.x * blockDim.x);

  /*---------------------------------------- kernel body -------------------------------------------*/

  // load data from global memory to registers
  if (tid < nn) reg_avg_foo = g_avg_foo[tid];

  // normalize avg foo
  if (tid < nn) reg_avg_foo /= norm_cst;
  __syncthreads();

  // store data in global memory
  if (tid < nn) g_avg_foo[tid] = reg_avg_foo ;

  return;
}

__global__ void particle2df(double *g_avg_ddf, int n_bin_ddf, double L, double *g_avg_vdf, int n_vdf,
                            int n_bin_vdf, double vmax, double vmin, particle *g_p, int num_p)
{
  /*--------------------------------------- kernel variables ---------------------------------------*/

  // kernel shared memory
  int *sh_ddf = (int *) sh_mem;                  // shared density distribution function
  int *sh_vdf = &sh_ddf[n_bin_ddf];              // shared velocity distribution functions (vdf)
```

```
441   int *sh_num_p_vdf = &sh_vdf[n_bin_vdf*n_vdf]; // shared number of partilces in each vdf
442
443   // kernel registers
444   particle reg_p;
445   int bin_index;
446   int vdf_index;
447   double bin_size;
448
449   int tidx = (int) threadIdx.x;
450   int bdim = (int) blockDim.x;
451   int tid = (int) (threadIdx.x + blockIdx.x * blockDim.x);
452
453   /*-------------------------------------------- kernel body -------------------------------------------*/
454
455   // initialize shared memory
456   for (int i = tidx; i < n_bin_ddf+(n_bin_vdf+1)*n_vdf; i+=bdim) sh_ddf[i] = 0;
457   __syncthreads();
458
459   // analize particles
460   if (tid < num_p) {
461     // load particle data from global memory to registers
462     reg_p = g_p[tid];
463
464     // add information to shared density distribution functions
465     bin_size = L/n_bin_ddf;
466     bin_index = __double2int_rd(reg_p.r/bin_size);
467     atomicAdd(&sh_ddf[bin_index], 1);
468
469     // add information to shared velocity distribution function
470     bin_size = L/n_vdf;
471     vdf_index = __double2int_rd(reg_p.r/bin_size);
472     bin_size = (vmax-vmin)/double(n_bin_vdf);
473     bin_index = __double2int_rd((reg_p.v-vmin)/bin_size);
474     if (bin_index < 0) {
475       bin_index = 0;
476     } else if (bin_index >= n_bin_vdf) {
477       bin_index = n_bin_vdf-1;
478     }
479     atomicAdd(&sh_vdf[bin_index+vdf_index*n_bin_vdf], 1);
480     atomicAdd(&sh_num_p_vdf[vdf_index], 1);
481   }
482
483   // syncronize threads to wait until all particles have been analized
484   __syncthreads();
485
486   // normalize density distribution function and add it to global averaged one
487   for (int i = tidx; i < n_bin_ddf; i += bdim) {
488     atomicAdd(&g_avg_ddf[i], double(sh_ddf[i])/double(num_p));
489   }
490   __syncthreads();
491
492   // normalize velocity distribution functions and add them to global averaged ones
493   for (int i = tidx; i < n_vdf*n_bin_vdf; i += bdim) {
494     if (sh_num_p_vdf[i/n_bin_vdf] != 0) {
495       atomicAdd(&g_avg_vdf[i], double(sh_vdf[i])/double(sh_num_p_vdf[i/n_bin_vdf]));
496     }
497   }
498
499   return;
500 }
501
502 __global__ void energy_kernel(double *g_U, double *g_phi, int nn, double ds, particle *g_p, double m,
503                               double q, int num_p)
504 {
505   /*------------------------------------------ kernel variables -----------------------------------------*/
506
507   // kernel shared memory
508   double *sh_phi = (double *) sh_mem;   // mesh potential
509   double *sh_U = &sh_phi[nn];           // acumulation of energy in each block
510
511   // kernel registers
512   int tid = (int) (threadIdx.x + blockIdx.x * blockDim.x);
513   int tidx = (int) threadIdx.x;
514   int bid = (int) blockIdx.x;
515   int bdim = (int) blockDim.x;
516
517   int ic;
518   double dist;
519
520   particle reg_p;
521
522   /*-------------------------------------------- kernel body -------------------------------------------*/
523
524   // load potential data from global to shared memory
525   for (int i = tidx; i < nn; i += bdim) {
526     sh_phi[i] = g_phi[i];
527   }
```

```
528
529   // initialize energy acumulation's variables
530   sh_U[tidx] = 0.0;
531   __syncthreads();
532
533   // analize energy of each particle
534   if (tid < num_p) {
535     // load particle in registers
536     reg_p = g_p[tid];
537     // calculate what cell the particle is in
538     ic = __double2int_rd(reg_p.r/ds);
539     // calculate distances from particle to down vertex of the cell
540     dist = fabs(__int2double_rn(ic)*ds-reg_p.r)/ds;
541     // evaluate potential energy of particle
542     sh_U[tidx] = (sh_phi[ic]*(1.0-dist)+sh_phi[ic+1]*dist)*q;
543     // evaluate kinetic energy of particle
544     sh_U[tidx] += 0.5*m*reg_p.v*reg_p.v;
545   }
546   __syncthreads();
547
548   // reduction for obtaining total energy in current block
549   for (int stride = 1; stride < bdim; stride *= 2) {
550     if ((tidx%(stride*2) == 0) && (tidx+stride < bdim)) {
551       sh_U[tidx] += sh_U[tidx+stride];
552     }
553     __syncthreads();
554   }
555
556   // store total energy of current block in global memory
557   if (tidx == 0) g_U[bid] = sh_U[0];
558
559   return;
560 }
```

Code B.10: CUPIC1D1V_PP source file diagnostic.cu

```
1  /****************************************************************************************************
2   *                                                                                                  *
3   *    This file is part of CUPIC1D1V_PP, a code that simulates the interaction between a plasma and  *
4   *    a planar Langmuir probe in 1D using PIC techniques accelerated with the use of GPU hardware    *
5   *    (CUDA, extension of C/C++)                                                                    *
6   *                                                                                                  *
7   ****************************************************************************************************/
8
9  #ifndef DIAGNOSTIC_H
10 #define DIAGNOSTIC_H
11
12 /********************************************* HEADERS ********************************************/
13
14 #include "stdh.h"
15 #include "init.h"
16 #include "cuda.h"
17
18 /************************************** SIMBOLIC CONSTANTS ****************************************/
19
20 #define AVG_MESH_BLOCK_DIM 512      // block dimension for mesh_sum and mesh_norm
21 #define ENERGY_BLOCK_DIM 512        // block dimension for energy solver kernel
22 #define PARTICLE2DF_BLOCK_DIM 512   // block dimension for particle2df kernel
23
24 /************************************** FUNCTION PROTOTIPES ***************************************/
25
26 // host function
27 void avg_mesh(double *d_foo, double *d_avg_foo, int *count);
28 void eval_df(double *d_avg_ddf, double *d_avg_vdf, double vmax, double vmin, particle *d_p, int num_p,
29              int *count);
30 double eval_particle_energy(double *d_phi,  particle *d_p, double m, double q, int num_p);
31 void particles_snapshot(particle *d_p, int num_p, string filename);
32 void save_mesh(double *d_m, string filename);
33 void save_ddf(double *d_avg_ddf, string filename);
34 void save_vdf(double *d_avg_vdf, double vmax, double vmin, string filename);
35 void save_log(double t, int num_e, int num_i, double U_e, double U_i, double vd_e, double vd_i,
36               double *d_phi);
37 //double calculate_vd_i(double dtin_i);
38
39 // device kernels
40 __global__ void mesh_sum(double *g_foo, double *g_avg_foo, int nn);
41 __global__ void mesh_norm(double *g_avg_foo, double norm_cst, int nn);
42 __global__ void particle2df(double *g_avg_ddf, int n_bin_ddf, double L, double *g_avg_vdf, int n_vdf,
43                             int n_bin_vdf, double vmax, double vmin, particle *g_p, int num_p);
44 __global__ void energy_kernel(double *g_U, double *g_phi, int nn, double ds, particle *g_p, double m,
45                               double q, int num_p);
46
47 // device functions
48
49 #endif
```

Code B.11: CUPIC1D1V_PP source file diagnostic.h

B.7. CUDA module

This module contains a few functions related to the use of the GPU including CUDA errors handling and intrinsic function definitions. (sources: cuda.cu, cuda.h)

```
/****************************************************************************************************
 *                                                                                                  *
 *    This file is part of CUPIC1D1V_PP, a code that simulates the interaction between a plasma and *
 *    a planar Langmuir probe in 1D using PIC techniques accelerated with the use of GPU hardware   *
 *    (CUDA, extension of C/C++)                                                                    *
 *                                                                                                  *
 ****************************************************************************************************/

/********************************************* HEADERS *********************************************/

#include "cuda.h"

/*********************************** HOST FUNCTION DEFINITIONS ************************************/

void cu_check(cudaError_t cuError, const string file, const int line)
{
  /*------------------------------------- function variables --------------------------------------*/

  /*--------------------------------------- function body -----------------------------------------*/

  if (0 == cuError)
  {
    return;
  } else
  {
    cout << "CUDA␣error␣found␣in␣file␣" << file << "␣at␣line␣" << line << ".␣(error␣code:␣"
         << cuError << ")" << endl;
    cout << "Exiting␣simulation" << endl;
    exit(1);
  }
}

void cu_sync_check(const string file, const int line)
{
  /*------------------------------------- function variables --------------------------------------*/
  cudaError_t cuError;

  /*--------------------------------------- function body -----------------------------------------*/

  cudaDeviceSynchronize();
  cuError = cudaGetLastError();
  if (0 == cuError)
  {
    return;
  } else
  {
    cout << "CUDA␣error␣found␣in␣file␣" << file << "␣at␣line␣" << line << ".␣(error␣code:␣" << cuError
         << ")" << endl;
    cout << "Exiting␣simulation" << endl;
    exit(1);
  }
}

/********************************** DEVICE FUNCTION DEFINITIONS ***********************************/

__device__ double atomicAdd(double* address, double val)
{
  /*------------------------------------- function variables --------------------------------------*/
  unsigned long long int* address_as_ull = (unsigned long long int*)address;
  unsigned long long int old = *address_as_ull, assumed;

  /*--------------------------------------- function body -----------------------------------------*/
  do
  {
    assumed = old;
    old = atomicCAS(address_as_ull, assumed, __double_as_longlong(val+__longlong_as_double(assumed)));
  } while (assumed != old);

  return __longlong_as_double(old);
}

__device__ double atomicSub(double* address, double val)
{
  /*------------------------------------- function variables --------------------------------------*/
  unsigned long long int* address_as_ull = (unsigned long long int*)address;
  unsigned long long int old = *address_as_ull, assumed;

  /*--------------------------------------- function body -----------------------------------------*/
  do
  {
    assumed = old;
```

```
    old = atomicCAS(address_as_ull, assumed, __double_as_longlong(val-__longlong_as_double(assumed)));
  } while (assumed != old);

  return __longlong_as_double(old);
}
```

Code B.12: CUPIC1D1V_PP source file cuda.cu

```
/****************************************************************************************************
 *                                                                                                  *
 *    This file is part of CUPIC1D1V_PP, a code that simulates the interaction between a plasma and  *
 *    a planar Langmuir probe in 1D using PIC techniques accelerated with the use of GPU hardware    *
 *    (CUDA, extension of C/C++)                                                                    *
 *                                                                                                  *
 ****************************************************************************************************/

#ifndef CUDA_H
#define CUDA_H

/********************************************* HEADERS *********************************************/

#include "stdh.h"

/*************************************** SIMBOLIC CONSTANTS ***************************************/

/*************************************** FUNCTION PROTOTIPES **************************************/
// host function
void cu_check(cudaError_t cuError, const string file, const int line);
void cu_sync_check(const string file, const int line);

// device kernels


// device functions (overload atomic functions for double precision support)
__device__ double atomicAdd(double* address, double val);
__device__ double atomicSub(double* address, double val);

#endif
```

Code B.13: CUPIC1D1V_PP source file cuda.h

B.8. Extra headers

Extra header files loaded in the previous modules. (sources: stdh.h, random.h, dynamic_sh_mem.h)

```
/****************************************************************************************************
 *                                                                                                  *
 *    This file is part of CUPIC1D1V_PP, a code that simulates the interaction between a plasma and  *
 *    a planar Langmuir probe in 1D using PIC techniques accelerated with the use of GPU hardware    *
 *    (CUDA, extension of C/C++)                                                                    *
 *                                                                                                  *
 ****************************************************************************************************/

#ifndef STD_H
#define STD_H

/********************************************* HEADERS *********************************************/

#include <stdlib.h>
#include <math.h>
#include <stdio.h>
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

/*************************************** SIMBOLIC CONSTANTS ***************************************/

#define PI 3.1415926535897932   //symbolic constant for PI

/*************************************** PARTICLE STRUCTURE ***************************************/

struct particle
{
  double r;
  double v;
};

#endif
```

Code B.14: CUPIC1D1V_PP source file stdh.h

```
/****************************************************************************************************
 *                                                                                                  *
 *    This file is part of CUPIC1D1V_PP, a code that simulates the interaction between a plasma and *
 *    a planar Langmuir probe in 1D using PIC techniques accelerated with the use of GPU hardware    *
 *    (CUDA, extension of C/C++)                                                                    *
 *                                                                                                  *
 ****************************************************************************************************/

#ifndef RAND_H
#define RAND_H

/********************************************* HEADERS **********************************************/

#include <curand_kernel.h>       //curand library for random number generation (__device__ functions)

/*************************************** SIMBOLIC CONSTANTS ****************************************/

#define CURAND_BLOCK_DIM 64      //block dimension for curand kernels

/*************************************** FUNCTION PROTOTIPES ***************************************/

#endif
```

Code B.15: CUPIC1D1V_PP source file random.h

```
/****************************************************************************************************
 *                                                                                                  *
 *    This file is part of CUPIC1D1V_PP, a code that simulates the interaction between a plasma and *
 *    a planar Langmuir probe in 1D using PIC techniques accelerated with the use of GPU hardware    *
 *    (CUDA, extension of C/C++)                                                                    *
 *                                                                                                  *
 ****************************************************************************************************/

#ifndef DYNAMIC_SH_MEM_H
#define DYNAMIC_SH_MEM_H

// variable for allowing dynamic allocation of __shared__ memory (used in several kernels)
extern __shared__ float sh_mem[];

#endif
```

Code B.16: CUPIC1D1V_PP source file dynamic_sh_mem.h

B.9. Additional files

File that automates the compilation process and input file to configure simulation parameters. (sources: makefile, input_data)

```
# Configuration

CC = g++
NVCC = nvcc
ARCHITECTURE = sm_20
NVCCFLAGS = -arch=$(ARCHITECTURE) #-Xptxas -v
LINKERFLAGS = -arch=$(ARCHITECTURE) -lcurand

OBJECTS = main.o init.o cc.o mesh.o particles.o diagnostic.o cuda.o


# Makefile orders

CUPIC : $(OBJECTS)
  $(NVCC) $(LINKERFLAGS) $(OBJECTS) -o cupic
  rm -f *~
  mv ./cupic ../bin/cupic

main.o : main.cu
  $(NVCC) $(NVCCFLAGS) -dc main.cu -o main.o

init.o : init.cu init.h
  $(NVCC) $(NVCCFLAGS) -dc init.cu -o init.o

cc.o : cc.cu cc.h
  $(NVCC) $(NVCCFLAGS) -dc cc.cu -o cc.o

mesh.o : mesh.cu mesh.h
  $(NVCC) $(NVCCFLAGS) -dc mesh.cu -o mesh.o

particles.o : particles.cu particles.h
  $(NVCC) $(NVCCFLAGS) -dc particles.cu -o particles.o

```

```
34 diagnostic.o : diagnostic.cu diagnostic.h
35   $(NVCC) $(NVCCFLAGS) -dc diagnostic.cu -o diagnostic.o
36
37 cuda.o : cuda.cu cuda.h
38   $(NVCC) $(NVCCFLAGS) -dc cuda.cu -o cuda.o
39
40 .PHONY : clean lines
41
42 clean :
43   rm -f *.o *~
44   clear
45
46 lines :
47   git ls-files | xargs wc -l
```

Code B.17: CUPIC1D1V_PP compilation file makefile

```
1  #execution configuration
2  n_ini = 0;
3  n_prev = 0;
4  n_save = 1000;
5  n_fin = 10000000;
6  #plasma properties
7  ne = 1.0e9;
8  Te = 1.0e3;
9  beta = 0.0e-2;
10 vd_e = 0.0;
11 vd_i = -0.02;
12 gamma = 1.0e3;
13 #probe properties
14 phi_p = -25.0e0;
15 #sizes of simulation
16 nc = 200;
17 ds = 2.0e-1;
18 dt = 1.0e-1;
19 #diagnostic properties
20 num_of_bins_ddf = 100;
21 num_of_vdf = 100;
22 num_of_bins_vdf = 100;
23 max_num_of_vth_e = 5.0;
24 min_num_of_vth_e = -5.0;
25 max_num_of_vth_i = 0.0;
26 min_num_of_vth_i = -0.3;
27 #calibration configuration
28 ion_current_calibration = 1;
29 #floating potential configuration
30 floating_potential = 0;
```

Code B.18: CUPIC1D1V_PP input file input_data

Appendix C

CUPIC1D2V_CP sources

This appendix is devoted to the source code of our simulation of the contact of a cylindrical Langmuir probe with a plasma. The code is divided into seven modules, each one taking care of an specific task. Also, there are a few extra header files that are loaded from the previous modules whenever they are needed, and a makefile that takes care of the compilation of the different modules to produce the simulation binary (makefile).

In the following sections the source files of the different modules are shown.

C.1. Main module

This is the main module of the simulation, it handles the simulation by calling functions that belongs to the rest of the modules. (sources: main.cu)

```
/****************************************************************************************************
 *                                                                                                  *
 *    This file is part of CUPIC1D2V_CP, a code that simulates the interaction between a plasma and *
 *    a cylindrical Langmuir probe in 1D using PIC techniques accelerated with the use of GPU       *
 *    hardware (CUDA, extension of C/C++)                                                           *
 *                                                                                                  *
 ****************************************************************************************************/

/********************************************* HEADERS *********************************************/

#include "stdh.h"
#include "init.h"
#include "cc.h"
#include "mesh.h"
#include "particles.h"
#include "diagnostic.h"

/************************** MAIN FUNCTION ***************************/

int main (int argc, const char* argv[])
{
  /*--------------------------- function variables -----------------------*/

  // host variables definition
  double t;                             // time of simulation
  const double dt = init_dt();          // time step
  const int n_ini = init_n_ini();       // number of first iteration
  const int n_prev = init_n_prev();     // number of iterations before start analizing
  const int n_save = init_n_save();     // number of iterations between diagnostics
  const int n_fin = init_n_fin();       // number of last iteration
  int num_i;                            // number of particles (electrons and ions)
  int nn = init_nn();                   // number of nodes
  double U_i;                           // system energy for electrons and ions
  double mi = init_mi();                // ion mass
  double dtin_i = init_dtin_i();        // time between ion insertions
  double q_pi = 0;                      // probe's positive acumulated charge (ions)
  double vd_i = init_vd_i();            // ion's drift velocity
  char filename[50];                    // filename for saved data

  ifstream ifile;
  ofstream ofile;

  // device variables definition
```

```
44    double *d_rho, *d_phi, *d_E;              // mesh properties
45    double *d_avg_rho, *d_avg_phi, *d_avg_E;  // mesh averaged properties
46    double *d_avg_ddf_i, *d_avg_vdf_i;        // density and velocity distribution function for ions
47    double v_max_i = init_v_max_i();          // maximun velocity of ions (for histograms)
48    double v_min_i = init_v_min_i();          // minimun velocity of ions (for histograms)
49    int count_df_i = 0;                       // |
50    int count_rho = 0;                        // |-> counters for avg data
51    int count_phi = 0;                        // |
52    int count_E = 0;                          // |
53    particle *d_i;                            // particles vectors
54    curandStatePhilox4_32_10_t *state;        // philox state for __device__ random number generation
55
56    /*----------------------------- function body -------------------------*/
57
58    //---- INITIALITATION OF SIMULATION
59
60    // initialize device and simulation variables
61    init_dev();
62    init_sim(&d_rho, &d_phi, &d_E, &d_avg_rho, &d_avg_phi, &d_avg_E, &d_i, &num_i, &d_avg_ddf_i,
63             &d_avg_vdf_i, &t, &state);
64
65    // save initial state
66    sprintf(filename, "../output/particles/ions_t_%d", n_ini);
67    particles_snapshot(d_i, num_i, filename);
68    sprintf(filename, "../output/charge/avg_charge_t_%d", n_ini);
69    save_mesh(d_avg_rho, filename);
70    sprintf(filename, "../output/potential/avg_potential_t_%d", n_ini);
71    save_mesh(d_avg_phi, filename);
72    sprintf(filename, "../output/field/avg_field_t_%d", n_ini);
73    save_mesh(d_avg_E, filename);
74    t += dt;
75
76    //---- SIMULATION BODY
77
78    for (int i = n_ini+1; i <= n_fin; i++, t += dt) {
79      // simulate one time step
80      charge_deposition(d_rho, d_phi, d_i, num_i);
81      poisson_solver(1.0e-4, d_rho, d_phi);
82      field_solver(d_phi, d_E);
83      particle_mover(d_i, num_i, d_E);
84      cc(t, &num_i, &d_i, &dtin_i, &vd_i, &q_pi, d_phi, d_E, state);
85
86      // average mesh variables and distribution functions
87      avg_mesh(d_rho, d_avg_rho, &count_rho);
88      avg_mesh(d_phi, d_avg_phi, &count_phi);
89      avg_mesh(d_E, d_avg_E, &count_E);
90      eval_df(d_avg_ddf_i, d_avg_vdf_i, v_max_i, v_min_i, d_i, num_i, &count_df_i);
91
92      // store data
93      if (i>=n_prev && i%n_save==0) {
94        // save particles (snapshot)
95        sprintf(filename, "../output/particles/ions_t_%d", i);
96        particles_snapshot(d_i, num_i, filename);
97
98        // save mesh properties
99        sprintf(filename, "../output/charge/avg_charge_t_%d", i);
100       save_mesh(d_avg_rho, filename);
101       sprintf(filename, "../output/potential/avg_potential_t_%d", i);
102       save_mesh(d_avg_phi, filename);
103       sprintf(filename, "../output/field/avg_field_t_%d", i);
104       save_mesh(d_avg_E, filename);
105
106       // save distribution functions
107       sprintf(filename, "../output/particles/ions_ddf_t_%d", i);
108       save_ddf(d_avg_ddf_i, filename);
109       sprintf(filename, "../output/particles/ions_vdf_t_%d", i);
110       save_vdf(d_avg_vdf_i, v_max_i, v_min_i, filename);
111
112       // save log
113       U_i = eval_particle_energy(d_phi,  d_i, mi, 1.0, num_i);
114       save_log(t, num_i, U_i, &q_pi, vd_i, d_phi);
115
116       cout << "iteration␣=␣" << i << endl;
117     }
118   }
119
120   //---- END OF SIMULATION
121
122   cout << "Simulation␣finished!" << endl;
123   return 0;
124 }
```

Code C.1: CUPIC1D2V_CP source file main.cu

C.2. Initialisation module

This is the module that handles the initialisation of the different variables of the simulation. It also prescribes the initial conditions for the system. (sources: init.cu, init.h)

```
/****************************************************************************************************
 *                                                                                                  *
 *    This file is part of CUPIC1D2V_CP, a code that simulates the interaction between a plasma and  *
 *    a cylindrical Langmuir probe in 1D using PIC techniques accelerated with the use of GPU        *
 *    hardware (CUDA, extension of C/C++)                                                            *
 *                                                                                                  *
 ****************************************************************************************************/

/********************************************** HEADERS ********************************************/

#include "init.h"

/********************************** HOST FUNCTION DEFINITIONS *************************************/

void init_dev(void)
{
  /*------------------------------------ function variables ---------------------------------------*/
  // host memory
  int dev;
  int devcnt;
  cudaDeviceProp devProp;
  cudaError_t cuError;

  // device memory

  /*-------------------------------------- function body ------------------------------------------*/

  // check for devices instaled in the host
  cuError = cudaGetDeviceCount(&devcnt);
  if (0 != cuError)
  {
    printf("Cuda error (%d) detected in 'init_dev(void)'\n", cuError);
    cout << "exiting simulation..." << endl;
    exit(1);
  }
  cout << devcnt << " devices present in the host:" << endl;
  for (dev = 0; dev < devcnt; dev++)
  {
    cudaGetDeviceProperties(&devProp, dev);
    cout << "  - Device " << dev << ":" << endl;
    cout << "    # " << devProp.name << endl;
    cout << "    # Compute capability " << devProp.major << "." << devProp.minor << endl;
  }

  // ask wich device to use
  cout << "Select in wich device simulation must be run: 0" << endl;
  dev = 0;  //cin >> dev;

  // set device to be used and reset it
  cudaSetDevice(dev);
  cudaDeviceReset();

  return;
}

void init_sim(double **d_rho, double **d_phi, double **d_E, double **d_avg_rho, double **d_avg_phi,
              double **d_avg_E, particle **d_i, int *num_i, double **d_avg_ddf_i, double **d_avg_vdf_i,
              double *t, curandStatePhilox4_32_10_t **state)
{
  /*------------------------------------ function variables ---------------------------------------*/
  // host memory
  const double dt = init_dt();
  const int n_ini = init_n_ini();

  // device memory

  /*-------------------------------------- function body ------------------------------------------*/

  cout << "n = " << init_n() << endl;
  // check if simulation start from initial condition or saved state
  if (n_ini == 0) {
    // adjust initial time
    *t = 0.;

    // create particles
    create_particles(d_i, num_i, state);

    // initialize mesh variables and their averaged counterparts
    initialize_mesh(d_rho, d_phi, d_E, *d_i, *num_i);

    // adjust velocities for leap-frog scheme
```

```
82      adjust_leap_frog(*d_i, *num_i, *d_E);
83
84      //initialize diagnostic variables
85      initialize_avg_mesh(d_avg_rho, d_avg_phi, d_avg_E);
86      initialize_avg_df(d_avg_ddf_i, d_avg_vdf_i);
87
88      cout << "Simulation␣initialized␣with␣" << *num_i << "␣particles." << endl << endl;
89    } else if (n_ini > 0) {
90      // adjust initial time
91      *t = n_ini*dt;
92
93      // read particle from file
94      load_particles(d_i, num_i, state);
95
96      // initialize mesh variables
97      initialize_mesh(d_rho, d_phi, d_E, *d_i, *num_i);
98
99      //initialize diagnostic variables
100     initialize_avg_mesh(d_avg_rho, d_avg_phi, d_avg_E);
101     initialize_avg_df(d_avg_ddf_i, d_avg_vdf_i);
102
103     cout << "Simulation␣state␣loaded␣from␣time␣t␣=␣" << *t << endl;
104   } else {
105     cout << "Wrong␣input␣parameter␣(n_ini<0)" << endl;
106     cout << "Stoppin␣simulation" << endl;
107     exit(1);
108   }
109
110   return;
111 }
112
113 void create_particles(particle **d_i, int *num_i, curandStatePhilox4_32_10_t **state)
114 {
115   /*-------------------------------------- function variables ---------------------------------------*/
116   // host memory
117   const double n = init_n();        // plasma density
118   const double me = init_me();      // electron's mass
119   const double mi = init_mi();      // ion's mass
120   const double kte = init_kte();    // electron's thermal energy
121   const double kti = init_kti();    // ion's thermal energy
122   const double vd_e = init_vd_e();  // electron's drift velocity
123   const double vd_i = init_vd_i();  // ion's drift velocity
124   const double L = init_L();        // size of simulation
125   const double ds = init_ds();      // spacial step
126
127   cudaError_t cuError;              // cuda error variable
128
129   // device memory
130
131   /*----------------------------------------- function body ------------------------------------------*/
132
133   // initialize curand philox states
134   cuError = cudaMalloc ((void **) state, CURAND_BLOCK_DIM*sizeof(curandStatePhilox4_32_10_t));
135   cu_check(cuError, __FILE__, __LINE__);
136   cudaGetLastError();
137   init_philox_state<<<1, CURAND_BLOCK_DIM>>>(*state);
138   cu_sync_check(__FILE__, __LINE__);
139
140   // calculate initial number of particles
141   *num_i = 0;
142
143   // allocate device memory for particle vectors
144   cuError = cudaMalloc ((void **) d_i, (*num_i)*sizeof(particle));
145   cu_check(cuError, __FILE__, __LINE__);
146
147   // create particles (ions)
148   cudaGetLastError();
149   create_particles_kernel<<<1, CURAND_BLOCK_DIM>>>(*d_i, *num_i, sqrt(kti/mi), vd_i, L, *state);
150   cu_sync_check(__FILE__, __LINE__);
151
152   return;
153 }
154
155 void initialize_mesh(double **d_rho, double **d_phi, double **d_E, particle *d_i, int num_i)
156 {
157   /*-------------------------------------- function variables ---------------------------------------*/
158   // host memory
159   const double phi_p = init_phi_p();                              // probe's potential
160   const double phi_s = -0.5*init_mi()*init_vd_i()*init_vd_i();    // sheath's edge potential
161   const int nn = init_nn();                                       // number of nodes
162   const int nc = init_nc();                                       // number of cells
163
164   double *h_phi;                                                  // host vector for potentials
165
166   cudaError_t cuError;                                            // cuda error variable
167
168   // device memory
```

```
169
170     /*---------------------------------------- function body -----------------------------------------*/
171
172     // allocate host memory for potential
173     h_phi = (double*) malloc(nn*sizeof(double));
174
175     // allocate device memory for mesh variables
176     cuError = cudaMalloc ((void **) d_rho, nn*sizeof(double));
177     cu_check(cuError, __FILE__, __LINE__);
178     cuError = cudaMalloc ((void **) d_phi, nn*sizeof(double));
179     cu_check(cuError, __FILE__, __LINE__);
180     cuError = cudaMalloc ((void **) d_E, nn*sizeof(double));
181     cu_check(cuError, __FILE__, __LINE__);
182
183     //initialize potential (host memory)
184     for (int i = 0; i < nn; i++)
185     {
186       h_phi[i] = phi_p + double(i)*(phi_s-phi_p)/double(nc);
187     }
188
189     // copy potential from host to device memory
190     cuError = cudaMemcpy (*d_phi, h_phi, nn*sizeof(double), cudaMemcpyHostToDevice);
191     cu_check(cuError, __FILE__, __LINE__);
192
193     // free host memory
194     free(h_phi);
195
196     // deposit charge into the mesh nodes
197     charge_deposition(*d_rho, *d_phi, d_i, num_i);
198
199     // solve poisson equation
200     poisson_solver(1.0e-4, *d_rho, *d_phi);
201
202     // derive electric fields from potential
203     field_solver(*d_phi, *d_E);
204
205     return;
206   }
207
208   void initialize_avg_mesh(double **d_avg_rho, double **d_avg_phi, double **d_avg_E)
209   {
210     /*------------------------------------- function variables --------------------------------------*/
211     // host memory
212     const int nn = init_nn();   // number of nodes
213
214     cudaError_t cuError;        // cuda error variable
215
216     // device memory
217
218     /*---------------------------------------- function body -----------------------------------------*/
219
220     // allocate device memory for averaged mesh variables
221     cuError = cudaMalloc ((void **) d_avg_rho, nn*sizeof(double));
222     cu_check(cuError, __FILE__, __LINE__);
223     cuError = cudaMalloc ((void **) d_avg_phi, nn*sizeof(double));
224     cu_check(cuError, __FILE__, __LINE__);
225     cuError = cudaMalloc ((void **) d_avg_E, nn*sizeof(double));
226     cu_check(cuError, __FILE__, __LINE__);
227
228     // initialize to zero averaged variables
229     cuError = cudaMemset ((void *) *d_avg_rho, 0, nn*sizeof(double));
230     cu_check(cuError, __FILE__, __LINE__);
231     cuError = cudaMemset ((void *) *d_avg_phi, 0, nn*sizeof(double));
232     cu_check(cuError, __FILE__, __LINE__);
233     cuError = cudaMemset ((void *) *d_avg_E, 0, nn*sizeof(double));
234     cu_check(cuError, __FILE__, __LINE__);
235
236     return;
237   }
238
239   void initialize_avg_df(double **d_avg_ddf_i, double **d_avg_vdf_i)
240   {
241     /*------------------------------------- function variables --------------------------------------*/
242     // host memory
243     const int n_bin_ddf = init_n_bin_ddf();   // number of bins for density distribution function
244     const int n_bin_vdf = init_n_bin_vdf();   // number of bins for velocity distribution function
245     const int n_vdf = init_n_vdf();           // number of velocity distribution functions to calculate
246
247     cudaError_t cuError;        // cuda error variable
248
249     // device memory
250
251     /*---------------------------------------- function body -----------------------------------------*/
252
253     // allocate device memory for averaged distribution functions
254     cuError = cudaMalloc ((void **) d_avg_ddf_i, n_bin_ddf*sizeof(double));
255     cu_check(cuError, __FILE__, __LINE__);
```

```
256   cuError = cudaMalloc ((void **) d_avg_vdf_i, n_bin_vdf*n_vdf*sizeof(double));
257   cu_check(cuError, __FILE__, __LINE__);
258 
259   // initialize to zero averaged distribution functions
260   cuError = cudaMemset ((void *) *d_avg_ddf_i, 0, n_bin_ddf*sizeof(double));
261   cu_check(cuError, __FILE__, __LINE__);
262   cuError = cudaMemset ((void *) *d_avg_vdf_i, 0, n_bin_vdf*n_vdf*sizeof(double));
263   cu_check(cuError, __FILE__, __LINE__);
264 
265   return;
266 }
267 
268 void adjust_leap_frog(particle *d_i, int num_i, double *d_E)
269 {
270   /*------------------------------------- function variables --------------------------------------*/
271   // host memory
272   const double mi = init_mi();          // ion's mass
273   const double me = init_me();          // electron's mass
274   const double r_p = init_r_p();        // probe radius
275   const double ds = init_ds();          // spatial step size
276   const double dt = init_dt();          // temporal step size
277   const int nn = init_nn();             // number of nodes
278 
279   dim3 griddim, blockdim;               // kernel execution configurations
280   size_t sh_mem_size;                   // shared memory size
281 
282   // device memory
283 
284   /*--------------------------------------- function body -----------------------------------------*/
285 
286   // set grid and block dimensions for fix_velocity kernel
287   griddim = 1;
288   blockdim = PAR_MOV_BLOCK_DIM;
289 
290   // set shared memory size for fix_velocity kernel
291   sh_mem_size = nn*sizeof(double);
292 
293   // fix velocities (ions)
294   cudaGetLastError();
295   fix_velocity<<<griddim, blockdim, sh_mem_size>>>(1.0, mi, num_i, d_i, dt, ds, r_p, nn, d_E);
296   cu_sync_check(__FILE__, __LINE__);
297 
298   return;
299 }
300 
301 void load_particles(particle **d_i, int *num_i, curandStatePhilox4_32_10_t **state)
302 {
303   /*------------------------------------- function variables --------------------------------------*/
304   // host memory
305   char filename[50];
306 
307   cudaError_t cuError;              // cuda error variable
308 
309   // device memory
310 
311   /*--------------------------------------- function body -----------------------------------------*/
312 
313   // initialize curand philox states
314   cuError = cudaMalloc ((void **) state, CURAND_BLOCK_DIM*sizeof(curandStatePhilox4_32_10_t));
315   cu_check(cuError, __FILE__, __LINE__);
316   cudaGetLastError();
317   init_philox_state<<<1, CURAND_BLOCK_DIM>>>(*state);
318   cu_sync_check(__FILE__, __LINE__);
319 
320   // load particles
321   sprintf(filename, "./ions.dat");
322   read_particle_file(filename, d_i, num_i);
323 
324   return;
325 }
326 
327 void read_particle_file(string filename, particle **d_p, int *num_p)
328 {
329   /*------------------------------------- function variables --------------------------------------*/
330   // host memory
331   particle *h_p;                // host vector for particles
332 
333   ifstream myfile;              // file variables
334   char line[150];
335 
336   cudaError_t cuError;          // cuda error variable
337 
338   // device memory
339 
340   /*--------------------------------------- function body -----------------------------------------*/
341 
342   // get number of particles (test if n is correctly evaluated)
```

```
343   *num_p = 0;
344   myfile.open(filename.c_str());
345   if (myfile.is_open()) {
346     myfile.getline(line, 150);
347     while (!myfile.eof()) {
348       myfile.getline(line, 150);
349       *num_p += 1;
350     }
351   } else {
352     cout << "Error.␣Can't␣open␣" << filename << "␣file" << endl;
353   }
354   myfile.close();
355 
356   // allocate host and device memory for particles
357   h_p = (particle*) malloc(*num_p*sizeof(particle));
358   cuError = cudaMalloc ((void **) d_p, *num_p*sizeof(particle));
359   cu_check(cuError, __FILE__, __LINE__);
360 
361   // read particles from file and store in host memory
362   myfile.open(filename.c_str());
363   if (myfile.is_open()) {
364     myfile.getline(line, 150);
365     for (int i = 0; i<*num_p; i++) {
366       myfile.getline(line, 150);
367       sscanf (line, "␣%le␣%le␣%le␣\n", &h_p[i].r, &h_p[i].vr, &h_p[i].vt);
368     }
369   } else {
370     cout << "Error.␣Can't␣open␣" << filename << "␣file" << endl;
371   }
372   myfile.close();
373 
374   // copy particle vector from host to device memory
375   cuError = cudaMemcpy (*d_p, h_p, *num_p*sizeof(particle), cudaMemcpyHostToDevice);
376   cu_check(cuError, __FILE__, __LINE__);
377 
378   // free host memory
379   free(h_p);
380 
381   return;
382 }
383 
384 template <typename type> void read_input_file(type *data, int n)
385 {
386   /*--------------------------------------- function variables ---------------------------------------*/
387   // function variables
388   ifstream myfile;
389   char line[80];
390 
391   /*------------------------------------------ function body ------------------------------------------*/
392   myfile.open("../input/input_data");
393   if (myfile.is_open()) {
394     for (int i = 0; i < n; i++) myfile.getline(line, 80);
395     if (sizeof(type) == sizeof(int)) {
396       sscanf (line, "%*s␣=␣%d;\n", (int*) data);
397     } else if (sizeof(type) == sizeof(double)) {
398       sscanf (line, "%*s␣=␣%lf;\n", (double*) data);
399     }
400   } else {
401     cout << "Error.␣Input␣data␣file␣could␣not␣be␣opened" << endl;
402     exit(1);
403   }
404   myfile.close();
405 
406   return;
407 }
408 
409 int init_n_ini(void)
410 {
411   /*--------------------------------------- function variables ---------------------------------------*/
412   static int n_ini = -1;
413 
414   /*------------------------------------------ function body ------------------------------------------*/
415 
416   if (n_ini < 0) read_input_file(&n_ini, 2);
417 
418   return n_ini;
419 }
420 
421 int init_n_prev(void)
422 {
423   /*--------------------------------------- function variables ---------------------------------------*/
424   static int n_prev = -1;
425 
426   /*------------------------------------------ function body ------------------------------------------*/
427 
428   if (n_prev < 0) read_input_file(&n_prev, 3);
429 
```

```
430     return n_prev;
431   }
432
433   int init_n_save(void)
434   {
435     /*-------------------------------------- function variables ---------------------------------------*/
436     static int n_save = -1;
437
438     /*----------------------------------------- function body -----------------------------------------*/
439
440     if (n_save < 0) read_input_file(&n_save, 4);
441
442     return n_save;
443   }
444
445   int init_n_fin(void)
446   {
447     /*-------------------------------------- function variables ---------------------------------------*/
448     static int n_fin = -1;
449
450     /*----------------------------------------- function body -----------------------------------------*/
451
452     if (n_fin < 0) read_input_file(&n_fin, 5);
453
454     return n_fin;
455   }
456
457   double init_n(void)
458   {
459     /*-------------------------------------- function variables ---------------------------------------*/
460     const double Dl = init_Dl();
461     static double n = 0.0;
462
463     /*----------------------------------------- function body -----------------------------------------*/
464
465     if (n == 0.0) {
466       read_input_file(&n, 7);
467       n *= Dl*Dl*Dl;
468     }
469
470     return n;
471   }
472
473   double init_kte(void)
474   {
475     /*-------------------------------------- function variables ---------------------------------------*/
476
477     /*----------------------------------------- function body -----------------------------------------*/
478
479     return 1.0;
480   }
481
482   double init_kti(void)
483   {
484     /*-------------------------------------- function variables ---------------------------------------*/
485     static double beta = 0.0;
486
487     /*----------------------------------------- function body -----------------------------------------*/
488
489     if (beta == 0.0) read_input_file(&beta, 9);
490
491     return beta;
492   }
493
494   double init_vd_e(void)
495   {
496     /*-------------------------------------- function variables ---------------------------------------*/
497     static double vd_e = -1000.0;
498
499     /*----------------------------------------- function body -----------------------------------------*/
500
501     if (vd_e == -1000.0) read_input_file(&vd_e, 10);
502
503     return vd_e;
504   }
505
506   double init_vd_i(void)
507   {
508     /*-------------------------------------- function variables ---------------------------------------*/
509     static double vd_i = -1000.0;
510
511     /*----------------------------------------- function body -----------------------------------------*/
512
513     if (vd_i == -1000.0) read_input_file(&vd_i, 11);
514
515     return vd_i;
516   }
```

```
517
518 double init_me(void)
519 {
520   /*--------------------------------------- function variables ---------------------------------------*/
521
522   /*----------------------------------------- function body -------------------------------------------*/
523
524   return 1.0;
525 }
526
527 double init_mi(void)
528 {
529   /*--------------------------------------- function variables ---------------------------------------*/
530   static double gamma = 0.0;
531
532   /*----------------------------------------- function body -------------------------------------------*/
533
534   if (gamma == 0.0) read_input_file(&gamma, 12);
535
536   return gamma;
537 }
538
539 double init_qe(void)
540 {
541   /*--------------------------------------- function variables ---------------------------------------*/
542
543   /*----------------------------------------- function body -------------------------------------------*/
544
545   return -1.0;
546 }
547
548 double init_qi(void)
549 {
550   /*--------------------------------------- function variables ---------------------------------------*/
551
552   /*----------------------------------------- function body -------------------------------------------*/
553
554   return 1.0;
555 }
556
557 double init_r_p(void)
558 {
559   /*--------------------------------------- function variables ---------------------------------------*/
560   static double r_p = 0.0;
561
562   /*----------------------------------------- function body -------------------------------------------*/
563
564   if (r_p == 0.0) read_input_file(&r_p, 14);
565
566   return r_p;
567 }
568
569 double init_l_p(void)
570 {
571   /*--------------------------------------- function variables ---------------------------------------*/
572   static double l_p = 0.0;
573
574   /*----------------------------------------- function body -------------------------------------------*/
575
576   if (l_p == 0.0) read_input_file(&l_p, 15);
577
578   return l_p;
579 }
580
581 double init_theta_p(void)
582 {
583   /*--------------------------------------- function variables ---------------------------------------*/
584   static double theta_p = 0.0;
585
586   /*----------------------------------------- function body -------------------------------------------*/
587
588   if (theta_p == 0.0) read_input_file(&theta_p, 16);
589
590   return theta_p;
591 }
592
593 double init_phi_p(void)
594 {
595   /*--------------------------------------- function variables ---------------------------------------*/
596   static double phi_p = 0.0;
597
598   /*----------------------------------------- function body -------------------------------------------*/
599
600   if (phi_p == 0.0) read_input_file(&phi_p, 17);
601
602   return phi_p;
603 }
```

```
604
605 int init_nc(void)
606 {
607   /*------------------------------------- function variables -------------------------------------*/
608   static int nc = 0;
609
610   /*--------------------------------------- function body ----------------------------------------*/
611
612   if (nc == 0) read_input_file(&nc, 19);
613
614   return nc;
615 }
616
617 double init_ds(void)
618 {
619   /*------------------------------------- function variables -------------------------------------*/
620   static double ds = 0.0;
621
622   /*--------------------------------------- function body ----------------------------------------*/
623
624   if (ds == 0.0) read_input_file(&ds, 20);
625
626   return ds;
627 }
628
629 double init_dt(void)
630 {
631   /*------------------------------------- function variables -------------------------------------*/
632   static double dt = 0.0;
633
634   /*--------------------------------------- function body ----------------------------------------*/
635
636   if (dt == 0.0) read_input_file(&dt, 21);
637
638   return dt;
639 }
640
641 int init_n_bin_ddf(void)
642 {
643   /*------------------------------------- function variables -------------------------------------*/
644   static int n_bin_ddf = -1;
645
646   /*--------------------------------------- function body ----------------------------------------*/
647
648   if (n_bin_ddf < 0) read_input_file(&n_bin_ddf, 23);
649
650   return n_bin_ddf;
651 }
652
653 int init_n_vdf(void)
654 {
655   /*------------------------------------- function variables -------------------------------------*/
656   static int n_vdf = -1;
657
658   /*--------------------------------------- function body ----------------------------------------*/
659
660   if (n_vdf < 0) read_input_file(&n_vdf, 24);
661
662   return n_vdf;
663 }
664
665 int init_n_bin_vdf(void)
666 {
667   /*------------------------------------- function variables -------------------------------------*/
668   static int n_bin_vdf = -1;
669
670   /*--------------------------------------- function body ----------------------------------------*/
671
672   if (n_bin_vdf < 0) read_input_file(&n_bin_vdf, 25);
673
674   return n_bin_vdf;
675 }
676
677 double init_v_max_e(void)
678 {
679   /*------------------------------------- function variables -------------------------------------*/
680   static double v_max_e = 0;   // max velocity to consider in velocity histograms
681
682   /*--------------------------------------- function body ----------------------------------------*/
683
684   if (v_max_e == 0) read_input_file(&v_max_e, 26);
685
686   return v_max_e;
687 }
688
689 double init_v_min_e(void)
690 {
```

```
691   /*-------------------------------------- function variables ---------------------------------------*/
692   static double v_min_e = 0;   // min velocity to consider in velocity histograms
693
694   /*---------------------------------------- function body -----------------------------------------*/
695
696   if (v_min_e == 0) read_input_file(&v_min_e, 27);
697
698   return v_min_e;
699 }
700
701 double init_v_max_i(void)
702 {
703   /*-------------------------------------- function variables ---------------------------------------*/
704   static double v_max_i = 0;   // max velocity to consider in velocity histograms
705
706   /*---------------------------------------- function body -----------------------------------------*/
707
708   if (v_max_i == 0) read_input_file(&v_max_i, 28);
709
710   return v_max_i;
711 }
712
713 double init_v_min_i(void)
714 {
715   /*-------------------------------------- function variables ---------------------------------------*/
716   static double v_min_i = 0;   // min velocity to consider in velocity histograms
717
718   /*---------------------------------------- function body -----------------------------------------*/
719
720   if (v_min_i == 0) read_input_file(&v_min_i, 29);
721
722   return v_min_i;
723 }
724
725 bool floating_potential_is_on(void)
726 {
727   /*-------------------------------------- function variables ---------------------------------------*/
728   static int floating_potential_int = -1;
729
730   /*---------------------------------------- function body -----------------------------------------*/
731
732   if (floating_potential_int < 0) {
733     read_input_file(&floating_potential_int , 31);
734     if (floating_potential_int != 0 && floating_potential_int != 1) {
735       cout << "Found␣error␣in␣input_data␣file.␣Wrong␣floating_potential!\nStoping␣simulation.\n" <<
736               endl;
737       exit(1);
738     }
739   }
740
741   if (floating_potential_int == 1) return true;
742   else return false;
743 }
744
745 bool flux_calibration_is_on(void)
746 {
747   /*-------------------------------------- function variables ---------------------------------------*/
748   static int flux_calibration_is_on = -1;
749
750   /*---------------------------------------- function body -----------------------------------------*/
751
752   if (flux_calibration_is_on < 0) {
753     read_input_file(&flux_calibration_is_on , 33);
754     if (flux_calibration_is_on != 0 && flux_calibration_is_on != 1) {
755       cout << "Found␣error␣in␣input_data␣file.␣Wrong␣flux_calibration_is_on␣!\nStoping␣simulation.\n"
756            << endl;
757       exit(1);
758     }
759   }
760
761   if (flux_calibration_is_on == 1) return true;
762   else return false;
763 }
764
765 double init_increment(void)
766 {
767   /*-------------------------------------- function variables ---------------------------------------*/
768   static double increment = -1;
769
770   /*---------------------------------------- function body -----------------------------------------*/
771
772   if (increment < 0) {
773     read_input_file(&increment, 34);
774   }
775
776   return increment;
777 }
```

```
778
779 int init_avg_nodes(void)
780 {
781   /*-------------------------------------- function variables ---------------------------------------*/
782   static int avg_nodes = -1;
783
784   /*---------------------------------------- function body ------------------------------------------*/
785
786   if (avg_nodes < 0) {
787     read_input_file(&avg_nodes, 35);
788   }
789
790   return avg_nodes;
791 }
792
793 double init_field_tol(void)
794 {
795   /*-------------------------------------- function variables ---------------------------------------*/
796   static double tol = -1;
797
798   /*---------------------------------------- function body ------------------------------------------*/
799
800   if (tol < 0) {
801     read_input_file(&tol, 36);
802   }
803
804   return tol;
805 }
806
807 double init_dtin_e(void)
808 {
809   /*-------------------------------------- function variables ---------------------------------------*/
810   const double n = init_n();
811   const double l_p = init_l_p();
812   const double r_p = init_r_p();
813   const double theta = init_theta_p();
814   const double L = init_L();
815   const double me = init_me();
816   const double kte = init_kte();
817   const double vd_e = init_vd_e();
818   const double phi_s = -0.5*init_mi()*init_vd_i()*init_vd_i();
819   static double dtin_e = 0.0;
820
821   /*---------------------------------------- function body ------------------------------------------*/
822
823   if (dtin_e == 0.0) {
824     dtin_e = n*sqrt(kte/(2.0*PI*me))*exp(-0.5*me*vd_e*vd_e/kte);  // thermal component of input flux
825     dtin_e += 0.5*n*(-vd_e)*(1.0+erf(sqrt(0.5*me/kte)*(-vd_e)));  // drift component of input flux
826     dtin_e *= exp(phi_s);                                         // correction on density at sheath edge
827
828     dtin_e *= (r_p+L)*theta*l_p;      // number of particles that enter the simulation per unit of time
829     dtin_e = 1.0/dtin_e;              // time between consecutive particles injection
830   }
831
832   return dtin_e;
833 }
834
835 double init_dtin_i(void)
836 {
837   /*-------------------------------------- function variables ---------------------------------------*/
838   const double n = init_n();
839   const double l_p = init_l_p();
840   const double r_p = init_r_p();
841   const double theta = init_theta_p();
842   const double L = init_L();
843   const double mi = init_mi();
844   const double kti = init_kti();
845   const double vd_i = init_vd_i();
846   const double phi_s = -0.5*init_mi()*init_vd_i()*init_vd_i();
847   static double dtin_i = 0.0;
848
849   /*---------------------------------------- function body ------------------------------------------*/
850
851   if (dtin_i == 0.0) {
852     dtin_i = n*sqrt(kti/(2.0*PI*mi))*exp(-0.5*mi*vd_i*vd_i/kti);  // thermal component of input flux
853     dtin_i += 0.5*n*(-vd_i)*(1.0+erf(sqrt(0.5*mi/kti)*(-vd_i)));  // drift component of input flux
854     dtin_i *= exp(phi_s);                                         // density fix at sheath edge
855
856     dtin_i *= (r_p+L)*theta*l_p;      // number of particles that enter the simulation per unit of time
857     dtin_i = 1.0/dtin_i;              // time between consecutive particles injection
858   }
859
860   return dtin_i;
861 }
862
863 int init_nn(void)
864 {
```

```
865    /*--------------------------------------- function variables ----------------------------------------*/
866    static int nn = init_nc()+1;
867
868    /*----------------------------------------- function body -------------------------------------------*/
869
870    return nn;
871  }
872
873  double init_L(void)
874  {
875    /*--------------------------------------- function variables ----------------------------------------*/
876    static double L = init_ds() * (double) init_nc();
877
878    /*----------------------------------------- function body -------------------------------------------*/
879
880    return L;
881  }
882
883  double init_epsilon0(void)
884  {
885    /*--------------------------------------- function variables ----------------------------------------*/
886    double Te;
887    const double Dl = init_Dl();
888    static double epsilon0 = 0.0;
889
890    /*----------------------------------------- function body -------------------------------------------*/
891
892    if (epsilon0 == 0.0) {
893      read_input_file(&Te, 8);
894      epsilon0 = CST_EPSILON;          // SI units
895      epsilon0 *= CST_KB*Te;           // energy units
896      epsilon0 /= CST_E*CST_E;         // charge units
897      epsilon0 *= Dl;                  // length units
898    }
899
900    return epsilon0;
901  }
902
903  double init_Dl(void)
904  {
905    /*--------------------------------------- function variables ----------------------------------------*/
906    double ne, Te;
907    static double Dl = 0.0;
908
909    /*----------------------------------------- function body -------------------------------------------*/
910
911    if (Dl == 0.0) {
912      read_input_file(&ne, 7);
913      read_input_file(&Te, 8);
914      Dl = sqrt(CST_EPSILON*CST_KB*Te/(ne*CST_E*CST_E));
915    }
916
917    return Dl;
918  }
919
920  double init_vth_e(void)
921  {
922    /*--------------------------------------- function variables ----------------------------------------*/
923    static double kte = init_kte();          // thermal energy of electrons
924    static double me = init_me();            // electron mass
925    static double vth_e = sqrt(kte/me);      // thermal velocity of electrons
926
927    /*----------------------------------------- function body -------------------------------------------*/
928
929    return vth_e;
930  }
931
932  double init_vth_i(void)
933  {
934    /*--------------------------------------- function variables ----------------------------------------*/
935    static double kti = init_kti();          // thermal energy of ions
936    static double mi = init_mi();            // ion mass
937    static double vth_i = sqrt(kti/mi);      // thermal velocity of ions
938
939    /*----------------------------------------- function body -------------------------------------------*/
940
941    return vth_i;
942  }
943
944  /************************************ DEVICE KERNELS DEFINITIONS ************************************/
945
946  __global__ void init_philox_state(curandStatePhilox4_32_10_t *state)
947  {
948    /*--------------------------------------- kernel variables ------------------------------------------*/
949    // kernel shared memory
950
951    // kernel registers
```

```
952    int tid = (int) threadIdx.x + (int) blockIdx.x * (int) blockDim.x;
953    curandStatePhilox4_32_10_t local_state;
954
955    /*---------------------------------------- kernel body ------------------------------------------*/
956
957    // load states in local memory
958    local_state = state[tid];
959
960    // initialize each thread state (seed, second seed, offset, pointer to state)
961    curand_init (0, tid, 0, &local_state);
962
963    // store initialized states in global memory
964    state[tid] = local_state;
965
966    return;
967  }
968
969  __global__ void create_particles_kernel(particle *g_p, int num_p, double sigma, double vd, double L,
970                                          curandStatePhilox4_32_10_t *state)
971  {
972    /*-------------------------------------- kernel variables ----------------------------------------*/
973    // kernel shared memory
974
975    // kernel registers
976    particle reg_p;
977    int tid = (int) threadIdx.x + (int) blockIdx.x * (int) blockDim.x;
978    int bdim = (int) blockDim.x;
979    curandStatePhilox4_32_10_t local_state;
980    double2 rnd;
981
982    /*---------------------------------------- kernel body ------------------------------------------*/
983
984    //---- load philox states from global memory
985    local_state = state[tid];
986
987    //---- create particles
988    for (int i = tid; i < num_p; i+=bdim) {
989      rnd.x = curand_uniform_double(&local_state);
990      reg_p.r = rnd.x*L;
991      rnd = curand_normal2_double(&local_state);
992      reg_p.vr = rnd.x*sigma+vd;
993      reg_p.vt = rnd.y*sigma;
994      // store particles in global memory
995      g_p[i] = reg_p;
996    }
997    __syncthreads();
998
999    //---- store philox states in global memory
1000   state[tid] = local_state;
1001
1002   return;
1003 }
1004
1005 __global__ void fix_velocity(double q, double m, int num_p, particle *g_p, double dt, double ds,
1006                              double r_p, int nn, double *g_E)
1007 {
1008   /*-------------------------------------- kernel variables ----------------------------------------*/
1009   // kernel shared memory
1010   double *sh_E = (double *) sh_mem;
1011
1012   // kernel registers
1013   int tid = (int) threadIdx.x;  // thread Id
1014   int bdim = (int) blockDim.x;  // block dimension
1015   particle reg_p;               // register particles
1016   int ic;                       // cell index
1017   double dist;                  // distance from particle to nearest down vertex (normalized to ds)
1018   double F;                     // force suffered for each register particle
1019
1020   /*---------------------------------------- kernel body ------------------------------------------*/
1021
1022   //---- load electric field in shared memory
1023   for (int i = tid; i<nn; i+=bdim) {
1024     sh_E[i] = g_E[i];
1025   }
1026   __syncthreads();
1027
1028   //---- load and analize and fix particles
1029   for (int i = tid; i<num_p; i += bdim) {
1030     // load particles from global to shared memory
1031     reg_p = g_p[i];
1032
1033     // analize particles
1034     ic = __double2int_rd(reg_p.r/ds);
1035
1036     // evaluate particle forces
1037     dist = fabs(reg_p.r-ic*ds)/ds;
1038     F = q*(sh_E[ic]*(1-dist)+sh_E[ic+1]*dist)+m*reg_p.vt*reg_p.vt/(reg_p.r+r_p);
```

```
1039
1040     // fix particle velocities
1041     reg_p.vr -= 0.5*dt*F/m;
1042
1043     // store back particles in global memory
1044     g_p[i] = reg_p;
1045   }
1046
1047   return;
1048 }
```

Code C.2: CUPIC1D2V_CP source file init.cu

```
1  /****************************************************************************************************
2   *                                                                                                  *
3   *    This file is part of CUPIC1D2V_CP, a code that simulates the interaction between a plasma and  *
4   *    a cylindrical Langmuir probe in 1D using PIC techniques accelerated with the use of GPU        *
5   *    hardware (CUDA, extension of C/C++)                                                           *
6   *                                                                                                  *
7   ****************************************************************************************************/
8
9  #ifndef INIT_H
10 #define INIT_H
11
12 /********************************************** HEADERS *********************************************/
13
14 #include "stdh.h"
15 #include "random.h"
16 #include "mesh.h"
17 #include "particles.h"
18 #include "dynamic_sh_mem.h"
19 #include "cuda.h"
20
21 /**************************************** SIMBOLIC CONSTANTS ****************************************/
22
23 #define CST_ME 9.109e-31      // electron mass (kg)
24 #define CST_E 1.602e-19       // electron charge (C)
25 #define CST_KB 1.381e-23      // boltzmann constant (m^2 kg s^-2 K^-1)
26 #define CST_EPSILON 8.854e-12 // free space electric permittivity (s^2 C^2 m^-3 kg^-1)
27
28 /*************************************** FUNCTION PROTOTIPES ****************************************/
29 // host functions
30 void init_dev(void);
31 void init_sim(double **d_rho, double **d_phi, double **d_E, double **d_avg_rho, double **d_avg_phi,
32               double **d_avg_E, particle **d_i, int *num_i, double **d_avg_ddf_i, double **d_avg_vdf_i,
33               double *t, curandStatePhilox4_32_10_t **state);
34 void create_particles(particle **d_i, int *num_i, curandStatePhilox4_32_10_t **state);
35 void initialize_mesh(double **d_rho, double **d_phi, double **d_E, particle *d_i, int num_i);
36 void initialize_avg_mesh(double **d_avg_rho, double **d_avg_phi, double **d_avg_E);
37 void initialize_avg_df(double **d_avg_ddf_i, double **d_avg_vdf_i);
38 void adjust_leap_frog(particle *d_i, int num_i, double *d_E);
39 void load_particles(particle **d_i, int *num_i, curandStatePhilox4_32_10_t **state);
40 void read_particle_file(string filename, particle **d_p, int *num_p);
41 template <typename type> void read_input_file(type *data, int n);
42 int init_n_ini(void);
43 int init_n_prev(void);
44 int init_n_save(void);
45 int init_n_fin(void);
46 double init_n(void);
47 double init_kte(void);
48 double init_kti(void);
49 double init_vd_e(void);
50 double init_vd_i(void);
51 double init_me(void);
52 double init_mi(void);
53 double init_qe(void);
54 double init_qi(void);
55 double init_r_p(void);
56 double init_l_p(void);
57 double init_theta_p(void);
58 double init_phi_p(void);
59 int init_nc(void);
60 double init_ds(void);
61 double init_dt(void);
62 int init_n_bin_ddf(void);
63 int init_n_vdf(void);
64 int init_n_bin_vdf(void);
65 double init_v_max_e(void);
66 double init_v_min_e(void);
67 double init_v_max_i(void);
68 double init_v_min_i(void);
69 bool floating_potential_is_on(void);
70 bool flux_calibration_is_on(void);
71 double init_increment(void);
72 int init_avg_nodes(void);
73 double init_field_tol(void);
74 double init_dtin_e(void);
```

```
double init_dtin_i(void);
int init_nn(void);
double init_L(void);
double init_epsilon0(void);
double init_Dl(void);
double init_vth_e(void);
double init_vth_i(void);

// device kernels
__global__ void init_philox_state(curandStatePhilox4_32_10_t *state);
__global__ void create_particles_kernel(particle *g_p, int num_p, double sigma, double vd, double L,
                                        curandStatePhilox4_32_10_t *state);
__global__ void fix_velocity(double q, double m, int num_p, particle *g_p, double dt, double ds,
                             double r_p, int nn, double *g_E);

#endif
```

Code C.3: `CUPIC1D2V_CP` source file `init.h`

C.3. Mesh module

This is the module in charge of all the mesh related algorithms, including: particle weighting, Poisson's equation solver, and field derivation. (sources: `mesh.cu`, `mesh.h`)

```
/****************************************************************************************************
 *                                                                                                  *
 *    This file is part of CUPIC1D2V_CP, a code that simulates the interaction between a plasma and  *
 *    a cylindrical Langmuir probe in 1D using PIC techniques accelerated with the use of GPU        *
 *    hardware (CUDA, extension of C/C++)                                                           *
 *                                                                                                  *
 ****************************************************************************************************/

/********************************************* HEADERS **********************************************/

#include "mesh.h"

/************************************ HOST FUNCTION DEFINITIONS *************************************/

void charge_deposition(double *d_rho, double *d_phi, particle *d_i, int num_i)
{
  /*------------------------------------ function variables ---------------------------------------*/
  // host memory
  static const double n = init_n();               // number density of particles at plasma
  static const double ds = init_ds();             // spatial step
  static const double l_p = init_l_p();           // lengh of cilindrical probe
  static const double r_p = init_r_p();           // probe radius
  static const double theta = init_theta_p();     // angular amplitude of probe
  static const int nn = init_nn();                // number of nodes

  dim3 griddim, blockdim;
  size_t sh_mem_size;
  cudaError_t cuError;

  // device memory

  /*--------------------------------------- function body -----------------------------------------*/

  // initialize device memory to zeros
  cuError = cudaMemset(d_rho, 0, nn*sizeof(double));
  cu_check(cuError, __FILE__, __LINE__);

  // set size of shared memory for particle_to_grid kernel
  sh_mem_size = nn*sizeof(double);

  // set dimensions of grid of blocks and blocks of threads for virtual_to_grid kernel
  blockdim = JACOBI_BLOCK_DIM;
  griddim = (int) ((nn-2)/JACOBI_BLOCK_DIM) + 1;

  // call to virtual_to_grid kernel (electrons)
  cudaGetLastError();
  virtual_to_grid<<<griddim, blockdim>>>(nn, ds, l_p, r_p, theta, d_rho, d_phi, n, -1.0);
  cu_sync_check(__FILE__, __LINE__);

  // set dimensions of grid of blocks and block of threads for particle_to_grid kernel (ions)
  blockdim = CHARGE_DEP_BLOCK_DIM;
  griddim = int(num_i/CHARGE_DEP_BLOCK_DIM)+1;

  // call to particle_to_grid kernel (ions)
  cudaGetLastError();
  particle_to_grid<<<griddim, blockdim, sh_mem_size>>>(nn, ds, l_p, r_p, theta, d_rho, d_i, num_i, 1.0);
  cu_sync_check(__FILE__, __LINE__);

```

```
59   return;
60 }
61
62 void poisson_solver(double max_error, double *d_rho, double *d_phi)
63 {
64   /*-------------------------------------- function variables ---------------------------------------*/
65   // host memory
66   static const double ds = init_ds();              // spatial step
67   static const int nn = init_nn();                 // number of nodes
68   static const double epsilon0 = init_epsilon0();  // electric permitivity of free space
69   static const double r_p = init_r_p();            // probe radius
70
71   double *h_error;
72   double t_error = max_error*10;
73   int min_iteration = 2*nn;
74
75   dim3 blockdim, griddim;
76   size_t sh_mem_size;
77   cudaError_t cuError;
78
79   // device memory
80   double *d_error;
81
82   /*----------------------------------------- function body -----------------------------------------*/
83
84   // set dimensions of grid of blocks and blocks of threads for jacobi kernel
85   blockdim = JACOBI_BLOCK_DIM;
86   griddim = (int) ((nn-2)/JACOBI_BLOCK_DIM) + 1;
87
88   // define size of shared memory for jacobi_iteration kernel
89   sh_mem_size = (2*JACOBI_BLOCK_DIM+2)*sizeof(double);
90
91   // allocate host and device memory for vector of errors
92   cuError = cudaMalloc((void **) &d_error, griddim.x*sizeof(double));
93   cu_check(cuError, __FILE__, __LINE__);
94   h_error = (double*) malloc(griddim.x*sizeof(double));
95
96   // execute jacobi iterations until solved
97   while(min_iteration>=0 || t_error>=max_error) {
98     // launch kernel for performing one jacobi iteration
99     cudaGetLastError();
100    jacobi_iteration<<<griddim, blockdim, sh_mem_size>>>(nn, ds, r_p, epsilon0, d_rho, d_phi, d_error);
101    cu_sync_check(__FILE__, __LINE__);
102
103    // copy error vector from  device to host memory
104    cuError = cudaMemcpy(h_error, d_error, griddim.x*sizeof(double), cudaMemcpyDeviceToHost);
105    cu_check(cuError, __FILE__, __LINE__);
106
107    // evaluate max error of the iteration
108    t_error = 0;
109    for (int i = 0; i<griddim.x; i++)
110    {
111      if (h_error[i] > t_error) t_error = h_error[i];
112    }
113
114    // actualize counter
115    min_iteration--;
116  }
117
118  // free device memory
119  cudaFree(d_error);
120  free(h_error);
121
122  return;
123 }
124
125 void field_solver(double *d_phi, double *d_E)
126 {
127   /*-------------------------- function variables ----------------------*/
128
129   // host memory
130   static const double ds = init_ds();   // spatial step
131   static const int nn = init_nn();      // number of nodes
132   dim3 blockdim, griddim;
133
134   // device memory
135
136   /*---------------------------- function body ------------------------*/
137
138   // set dimensions of grid of blocks and blocks of threads for jacobi kernel
139   blockdim = JACOBI_BLOCK_DIM;
140   griddim = (int) ((nn-2)/JACOBI_BLOCK_DIM) + 1;
141
142   // launch kernel for performing the derivation of the potential to obtain the electric field
143   cudaGetLastError();
144   field_derivation<<<griddim, blockdim>>>(nn, ds, d_phi, d_E);
145   cu_sync_check(__FILE__, __LINE__);
```

```
146
147     return;
148   }
149
150   /************************************ DEVICE KERNELS DEFINITIONS ************************************/
151
152   __global__ void particle_to_grid(int nn, double ds, double l_p, double r_p, double theta,
153                                    double *g_rho, particle *g_p, int num_p, double q)
154   {
155     /*-------------------------------------- kernel variables ---------------------------------------*/
156     // kernel shared memory
157     double *sh_partial_rho = (double *) sh_mem;   // partial rho of each bin
158
159     // kernel registers
160     int tidx = (int) threadIdx.x;
161     int tid = (int) (threadIdx.x + blockIdx.x*blockDim.x);
162     int bdim = (int) blockDim.x;
163     int ic;                        // cell index of each particle
164     particle reg_p;                // register copy of particle analized
165     double dist;                   // distance to down vertex of the cell
166
167     /*---------------------------------------- kernel body -----------------------------------------*/
168
169     //---- initialize shared memory variables
170
171     // initialize charge density in shared memory to 0.0
172     for (int i = tidx; i < nn; i+=bdim) {
173       sh_partial_rho[i] = 0.0;
174     }
175     __syncthreads();
176
177     //--- deposition of charge
178
179     if (tid < num_p) {
180       // load particle in registers
181       reg_p = g_p[tid];
182       // calculate what cell the particle is in
183       ic = __double2int_rd(reg_p.r/ds);
184       if (reg_p.r == (nn-1)*ds) ic = nn-2;
185       if (ic >= nn-1) printf("error␣2␣on␣tid␣=␣%d,␣ic␣=␣%d,␣p.r␣=␣%f\n", tidx, ic, reg_p.r);
186       // calculate distances from particle to down vertex of the cell
187       dist = fabs(__int2double_rn(ic)*ds-reg_p.r)/ds;
188       // acumulate charge in partial rho
189       atomicAdd(&sh_partial_rho[ic], q*(1.0-dist));    //down vertex
190       atomicAdd(&sh_partial_rho[ic+1], q*dist);        //upper vertex
191     }
192     __syncthreads();
193
194     //---- volume correction (shared memory)
195
196     for (int i = tidx+1; i < nn-1; i+=bdim) {
197       sh_partial_rho[i] /= l_p*ds*theta*(i*ds+r_p);
198     }
199     if (tidx == 0) {
200       sh_partial_rho[0] /= 0.5*l_p*theta*ds*(r_p+0.25*ds);
201       sh_partial_rho[nn-1] /= 0.5*l_p*theta*ds*(r_p+(nn-1.25)*ds);
202     }
203     __syncthreads();
204
205     //---- charge acumulation in global memory
206
207     for (int i = tidx; i < nn; i+=bdim) {
208       atomicAdd(&g_rho[i], sh_partial_rho[i]);
209     }
210     __syncthreads();
211
212     return;
213   }
214
215   __global__ void virtual_to_grid(int nn, double ds, double l_p, double r_p, double theta, double *g_rho,
216                                   double *g_phi, double n, double q)
217   {
218     /*-------------------------------------- kernel variables ---------------------------------------*/
219     // kernel shared memory
220
221     // kernel registers
222     double reg_phi, reg_rho;
223     int g_tid = (int) (threadIdx.x + blockDim.x * blockIdx.x);
224
225     /*---------------------------------------- kernel body -----------------------------------------*/
226
227     // load phi data from global to shared memory
228     if (g_tid < nn) reg_phi = g_phi[g_tid];
229
230     //--- deposition of charge
231     reg_rho = n*exp(reg_phi)*q;
232
```

```
  //---- store virtual charge in global memory
  if (g_tid < nn) g_rho[g_tid] = reg_rho;
  __syncthreads();

  return;
}


__global__ void jacobi_iteration (int nn, double ds, double r_p, double epsilon0, double *g_rho,
                                  double *g_phi, double *g_error)
{
  /*---------------------------------------- kernel variables ----------------------------------------*/
  // shared memory
  double *sh_old_phi= (double *) sh_mem;                          //
  double *sh_error = (double *) &sh_old_phi[JACOBI_BLOCK_DIM+2];   // manually set up shared memory

  // registers
  double new_phi, dummy_rho;
  int tid = (int) threadIdx.x;
  int sh_tid = (int) threadIdx.x + 1;
  int g_tid = (int) (threadIdx.x + blockDim.x * blockIdx.x) + 1;
  int bdim = (int) blockDim.x;
  int bid = (int) blockIdx.x;
  int gdim = (int) gridDim.x;

  /*------------------------------------------ kernel body -------------------------------------------*/

  // load phi data from global to shared memory
  if (g_tid < nn - 1) sh_old_phi[sh_tid] = g_phi[g_tid];

  // load comunication zones
  if (bid < gdim-1) {
    if (sh_tid == 1) sh_old_phi[sh_tid-1] = g_phi[g_tid-1];
    if (sh_tid == bdim) sh_old_phi[sh_tid+1] = g_phi[g_tid+1];
  } else {
    if (sh_tid == 1) sh_old_phi[sh_tid-1] = g_phi[g_tid-1];
    if (g_tid == nn-2) sh_old_phi[sh_tid+1] = g_phi[g_tid+1];
  }
  __syncthreads();

  // load charge density data into registers
  if (g_tid < nn - 1) dummy_rho = ds*ds*g_rho[g_tid]/epsilon0;
  __syncthreads();

  // actualize interior mesh points
  if (g_tid < nn - 1) new_phi = 0.5*(dummy_rho + sh_old_phi[sh_tid+1]*(1.0+ds/(2.0*(g_tid*ds+r_p))) +
                                sh_old_phi[sh_tid-1]*(1.0-ds/(2.0*(g_tid*ds+r_p))));
  __syncthreads();

  // store new values of phi in global memory
  if (g_tid < nn - 1) g_phi[g_tid] = new_phi;
  __syncthreads();

  // evaluate local errors
  if (g_tid < nn - 1) sh_error[tid] = fabs(new_phi-sh_old_phi[sh_tid]);
  __syncthreads();

  // reduction for obtaining maximum error in current block
  for (int stride = 1; stride < bdim; stride <<= 1) {
    if ((tid%(stride*2) == 0) && (tid+stride < bdim) && (g_tid+stride < nn-1)) {
      if (sh_error[tid]<sh_error[tid+stride]) sh_error[tid] = sh_error[tid+stride];
    }
    __syncthreads();
  }

  // store maximun error in global memory
  if (tid == 0) g_error[bid] = sh_error[tid];

  return;
}

__global__ void field_derivation (int nn, double ds, double *g_phi, double *g_E)
{
  /*---------------------------------------- kernel variables ----------------------------------------*/
  // shared memory
  __shared__ double sh_phi[JACOBI_BLOCK_DIM+2];

  // registers
  double reg_E;
  int sh_tid = (int) threadIdx.x + 1;
  int g_tid = (int) (threadIdx.x + blockDim.x * blockIdx.x) + 1;
  int bdim = (int) blockDim.x;
  int bid = (int) blockIdx.x;
  int gdim = (int) gridDim.x;

  /*------------------------------------------ kernel body -------------------------------------------*/

```

```
320    // load phi data from global to shared memory
321    if (g_tid < nn - 1) {
322      sh_phi[sh_tid] = g_phi[g_tid];
323    }
324    // load comunication zones
325    if (bid < gdim-1) {
326      if (sh_tid == 1) sh_phi[0] = g_phi[g_tid-1];
327      if (sh_tid == bdim) sh_phi[sh_tid+1] = g_phi[g_tid+1];
328    } else {
329      if (sh_tid == 1) sh_phi[sh_tid-1] = g_phi[g_tid-1];
330      if (g_tid == nn-1) sh_phi[sh_tid] = g_phi[g_tid];
331    }
332    __syncthreads();
333
334    // calculate electric fields in interior points
335    if (g_tid < nn - 1) {
336      reg_E = (sh_phi[sh_tid-1]-sh_phi[sh_tid+1])/(2.0*ds);
337    }
338    __syncthreads();
339
340    // store electric fields of interior points in global memory
341    if (g_tid < nn - 1) g_E[g_tid] = reg_E;
342
343    // calculate electric fields at proble and plasma
344    if (g_tid == nn-1) {
345      reg_E = (sh_phi[sh_tid-1]-sh_phi[sh_tid])/ds;
346      g_E[g_tid] = reg_E;
347    } else if (g_tid == 1) {
348      reg_E = (sh_phi[sh_tid-1]-sh_phi[sh_tid])/ds;
349      g_E[g_tid-1] = reg_E;
350    }
351
352    return;
353  }
```

Code C.4: `CUPIC1D2V_CP` source file `mesh.cu`

```
1   /****************************************************************************************************
2    *                                                                                                  *
3    *    This file is part of CUPIC1D2V_CP, a code that simulates the interaction between a plasma and  *
4    *    a cylindrical Langmuir probe in 1D using PIC techniques accelerated with the use of GPU        *
5    *    hardware (CUDA, extension of C/C++)                                                           *
6    *                                                                                                  *
7    ****************************************************************************************************/
8
9   #ifndef MESH_H
10  #define MESH_H
11
12  /********************************************** HEADERS ********************************************/
13
14  #include "stdh.h"
15  #include "init.h"
16  #include "dynamic_sh_mem.h"
17  #include "cuda.h"
18
19  /************************************** SIMBOLIC CONSTANTS *****************************************/
20
21  #define CHARGE_DEP_BLOCK_DIM 512   //block dimension for particle2grid kernel
22  #define JACOBI_BLOCK_DIM 128       //block dimension for jacovi_iteration kernel
23
24  /************************************** FUNCTION PROTOTIPES ****************************************/
25
26  // host function
27  void charge_deposition(double *d_rho, double *d_phi, particle *d_i, int num_i);
28  void poisson_solver(double max_error, double *d_rho, double *d_phi);
29  void field_solver(double *d_phi, double *d_E);
30
31  // device kernels
32  __global__ void particle_to_grid(int nn, double ds, double l_p, double r_p, double theta,
33                                  double *g_rho, particle *g_p, int num_p, double q);
34  __global__ void virtual_to_grid(int nn, double ds, double l_p, double r_p, double theta, double *g_rho,
35                                 double *g_phi, double n, double q);
36  __global__ void jacobi_iteration (int nn, double ds, double r_p, double epsilon0, double *g_rho,
37                                   double *g_phi, double *g_error);
38  __global__ void field_derivation (int nn, double ds, double *g_phi, double *g_E);
39
40  // device functions
41
42  #endif
```

Code C.5: `CUPIC1D2V_CP` source file `mesh.h`

C.4. Particles module

This is the module that manage the particles motion. It includes the field weighting algorithm as well as the particle mover, *i. e.* leap-frog algorithm. (sources: `particles.cu`, `particles.h`)

```
/****************************************************************************************************
 *                                                                                                  *
 *    This file is part of CUPIC1D2V_CP, a code that simulates the interaction between a plasma and  *
 *    a cylindrical Langmuir probe in 1D using PIC techniques accelerated with the use of GPU       *
 *    hardware (CUDA, extension of C/C++)                                                           *
 *                                                                                                  *
 ****************************************************************************************************/

/********************************************* HEADERS *********************************************/

#include "particles.h"

/*********************************** HOST FUNCTION DEFINITIONS ***********************************/

void particle_mover(particle *d_i, int num_i, double *d_E)
{
  /*-------------------------------------- function variables ---------------------------------------*/
  // host memory
  static const double me = init_me();     // electron's mass
  static const double mi = init_mi();     // ion's mass
  static const double qe = init_qe();     // electron's charge
  static const double qi = init_qi();     // ions's charge
  static const double ds = init_ds();     // spatial step
  static const double r_p = init_r_p();   // spatial step
  static const double dt = init_dt();     // time step
  static const int nn = init_nn();        // number of nodes

  dim3 griddim, blockdim;
  size_t sh_mem_size;

  // device memory

  /*----------------------------------------- function body ------------------------------------------*/

  // set size of __shared__ memory for leap_frog kernel
  sh_mem_size = nn*sizeof(double);

  //---- move ions

  // set dimensions of grid of blocks and blocks of threads for leap_frog kernel
  blockdim = PAR_MOV_BLOCK_DIM;
  griddim = int(num_i/PAR_MOV_BLOCK_DIM)+1;

  // call to leap_frog_step kernel (ions)
  cudaGetLastError();
  leap_frog_step<<<griddim, blockdim, sh_mem_size>>>(qi, mi, num_i, d_i, dt, ds, r_p, nn, d_E);
  cu_sync_check(__FILE__, __LINE__);

  return;
}

/*********************************** DEVICE KERNELS DEFINITIONS ***********************************/

__global__ void leap_frog_step(double q, double m, int num_p, particle *g_p, double dt, double ds,
                               double r_p, int nn, double *g_E)
{
  /*--------------------------------------- kernel variables ----------------------------------------*/
  // kernel shared memory
  double *sh_E = (double *) sh_mem;   // manually set up shared memory variables

  // kernel registers
  int tidx = (int) threadIdx.x;
  int tid = (int) threadIdx.x + (int) blockDim.x * (int) blockIdx.x;  // thread Id
  int bdim = (int) blockDim.x;  // block dimension
  particle reg_p;               // register particles
  int ic;                       // cell index
  double dist;                  // distance from particle to nearest down vertex (normalized to ds)
  double F;                     // force suffered for each register particle
  double dummy_r;               // intermediate new position

  /*------------------------------------------ kernel body ------------------------------------------*/

  //---- initialize shared memory variables

  // load fields from global memory
  for (int i = tidx; i<nn; i += bdim) {
    sh_E[i] = g_E[i];
  }
  __syncthreads();

  //---- Process batches of particles
```

```
82
83    if (tid < num_p) {
84      // load particle data in registers
85      reg_p = g_p[tid];
86
87      // find cell index
88      ic = __double2int_rd(reg_p.r/ds);
89
90      // evaluate distance to nearest down vertex (normalized to ds)
91      dist = fabs(reg_p.r-ic*ds)/ds;
92
93      // calculate particle's forces
94      F = q*(sh_E[ic]*(1.0-dist) + sh_E[ic+1]*dist)+m*reg_p.vt*reg_p.vt/(reg_p.r+r_p);
95
96      // move particles
97      reg_p.vr += dt*F/m;
98      dummy_r = reg_p.r + dt*reg_p.vr;
99      reg_p.vt *= (reg_p.r+r_p)/(dummy_r+r_p);
100     reg_p.r = dummy_r;
101
102     // store particle data in global memory
103     g_p[tid] = reg_p;
104   }
105
106   return;
107 }
```

Code C.6: CUPIC1D2V_CP source file particles.cu

```
1  /****************************************************************************************************
2   *                                                                                                  *
3   *    This file is part of CUPIC1D2V_CP, a code that simulates the interaction between a plasma and  *
4   *    a cylindrical Langmuir probe in 1D using PIC techniques accelerated with the use of GPU        *
5   *    hardware (CUDA, extension of C/C++)                                                           *
6   *                                                                                                  *
7   ****************************************************************************************************/
8
9  #ifndef PARTICLES_H
10 #define PARTICLES_H
11
12 /********************************************* HEADERS *********************************************/
13
14 #include "stdh.h"
15 #include "init.h"
16 #include "diagnostic.h"
17 #include "dynamic_sh_mem.h"
18 #include "cuda.h"
19
20 /*************************************** SIMBOLIC CONSTANTS ***************************************/
21
22 #define PAR_MOV_BLOCK_DIM 512   //block dimension for defragmentation kernel
23
24 /************************************** FUNCTION PROTOTIPES **************************************/
25
26 // host function
27 void particle_mover(particle *d_i, int num_i, double *d_E);
28
29 // device kernels
30 __global__ void leap_frog_step(double q, double m, int num_p, particle *g_p, double dt, double ds,
31                                double r_p, int nn, double *g_E);
32
33 // device functions
34
35 #endif
```

Code C.7: CUPIC1D2V_CP source file particles.h

C.5. Boundary conditions module

This is the module that takes care for the influx of particles coming from the plasma, as well as the absorption of particles at both boundaries. (sources: cc.cu, cc.h)

```
1  /****************************************************************************************************
2   *                                                                                                  *
3   *    This file is part of CUPIC1D2V_CP, a code that simulates the interaction between a plasma and  *
4   *    a cylindrical Langmuir probe in 1D using PIC techniques accelerated with the use of GPU        *
5   *    hardware (CUDA, extension of C/C++)                                                           *
6   *                                                                                                  *
7   ****************************************************************************************************/
8
9  /********************************************* HEADERS *********************************************/
```

```
10
11 #include "cc.h"
12
13 /************************************ HOST FUNCTION DEFINITIONS ************************************/
14
15 void cc (double t, int *num_i, particle **d_i, double *dtin_i, double *vd_i, double *q_pi,
16          double *d_phi, double *d_E, curandStatePhilox4_32_10_t *state)
17 {
18   /*-------------------------------------- function variables ---------------------------------------*/
19   // host memory
20   static const double me = init_me();                          //
21   static const double mi = init_mi();                          //
22   static const double kte = init_kte();                        // particle
23   static const double kti = init_kti();                        // properties
24   static const double vd_e = init_vd_e();                      //
25
26   static const double r_p = init_r_p();                        // probe radius
27   static const double theta = init_theta_p();                  // angular amplitude of the simulation
28   static const bool fp_is_on = floating_potential_is_on();     // probe is floating or not
29   static const bool flux_cal_on = flux_calibration_is_on(); // ion flux calibration is activated or not
30   static const int nc = init_nc();                             // number of cells
31   static const double ds = init_ds();                          // spatial step
32   static const double epsilon0 = init_epsilon0();              // epsilon0 in simulation units
33
34   static double tin_i = t+(*dtin_i);                           // time for next ion insertion
35
36   static double q_p = 0.0;                                     // net charge acumulated by the probe (not reseted)
37   double phi_s = -0.5*mi*(*vd_i)*(*vd_i);                      // potential at sheath edge
38   double dummy_phi_p;                                          // dummy probe potential
39
40   cudaError cuError;                                           // cuda error variable
41
42   // device memory
43
44   /*---------------------------------------- function body -----------------------------------------*/
45
46   //---- ions contour conditions
47   abs_emi_cc(t, &tin_i, *dtin_i, kti, mi, *vd_i, +1.0, q_pi, num_i, d_i, d_E, state);
48
49   //---- actualize probe potential because of the change in charge collected by the probe
50   if (fp_is_on) {
51     q_p += *q_pi;
52     dummy_phi_p = q_p/(2.0*theta*epsilon0*r_p);
53     if (dummy_phi_p > phi_s) dummy_phi_p = phi_s;
54     cuError = cudaMemcpy (&d_phi[0], &dummy_phi_p, sizeof(double), cudaMemcpyHostToDevice);
55     cu_check(cuError, __FILE__, __LINE__);
56   }
57
58   //---- actulize ion drift velocity if calibration is on
59   if (flux_cal_on) {
60     calibrate_ion_flux(vd_i, dtin_i, d_E, d_phi);
61   }
62
63   return;
64 }
65
66 void abs_emi_cc(double t, double *tin, double dtin, double kt, double m, double vd, double q,
67                 double *q_p, int *h_num_p, particle **d_p, double *d_E,
68                 curandStatePhilox4_32_10_t *state)
69 {
70   /*-------------------------------------- function variables ---------------------------------------*/
71   // host memory
72   static const double L = init_L();       //
73   static const double r_p = init_r_p();   // geometric properties
74   static const double ds = init_ds();     // of simulation
75   static const int nn = init_nn();        //
76
77   static const double dt = init_dt();     //
78   double fpt = t+dt;                      // timing variables
79   double fvt = t+0.5*dt;                  //
80
81   int in = 0;                             // number of particles added at plasma frontier
82   int h_num_abs_p;                        // host number of particles absorved at the probe
83
84   double dv;                              //
85   int i;                                  // variables for
86   double xmax, ymax, y1, y2;              // rejection method
87   double vth = sqrt(kt/m);                //
88
89   cudaError cuError;                      // cuda error variable
90   dim3 griddim, blockdim;                 // kernel execution configurations
91
92   // device memory
93   int *d_num_p;                           // device number of particles
94   int *d_num_abs_p;                       // device number of particles absorved at the probe
95   particle *d_dummy_p;                    // device dummy vector for particle storage
96
```

```
97  /*------------------------------------------ function body ------------------------------------------*/
98
99  // calculate number of particles that flow into the simulation
100 if((*tin) < fpt) in = 1 + int((fpt-(*tin))/dtin);
101
102 // copy number of particles from host to device
103 cuError = cudaMalloc((void **) &d_num_p, sizeof(int));
104 cu_check(cuError, __FILE__, __LINE__);
105 cuError = cudaMemcpy (d_num_p, h_num_p, sizeof(int), cudaMemcpyHostToDevice);
106 cu_check(cuError, __FILE__, __LINE__);
107
108 // initialize number of particles absorbed at the probe
109 cuError = cudaMalloc((void **) &d_num_abs_p, sizeof(int));
110 cu_check(cuError, __FILE__, __LINE__);
111 cuError = cudaMemset((void *) d_num_abs_p, 0, sizeof(int));
112 cu_check(cuError, __FILE__, __LINE__);
113
114 // execution configuration for particle remover kernel
115 griddim = 1;
116 blockdim = P_RMV_BLK_SZ;
117
118 // execute particle remover kernel
119 cudaGetLastError();
120 pRemover<<<griddim, blockdim>>>(*d_p, d_num_p, L, d_num_abs_p);
121 cu_sync_check(__FILE__, __LINE__);
122
123 // copy number of particles absorbed at the probe from device to host (and free device memory)
124 cuError = cudaMemcpy (&h_num_abs_p, d_num_abs_p, sizeof(int), cudaMemcpyDeviceToHost);
125 cu_check(cuError, __FILE__, __LINE__);
126 cuError = cudaFree(d_num_abs_p);
127 cu_check(cuError, __FILE__, __LINE__);
128
129 // actualize probe acumulated charge
130 *q_p += q*h_num_abs_p;
131
132 // copy new number of particles from device to host (and free device memory)
133 cuError = cudaMemcpy (h_num_p, d_num_p, sizeof(int), cudaMemcpyDeviceToHost);
134 cu_check(cuError, __FILE__, __LINE__);
135 cuError = cudaFree(d_num_p);
136 cu_check(cuError, __FILE__, __LINE__);
137
138 // resize of particle vector with new number of particles
139 cuError = cudaMalloc((void **) &d_dummy_p, ((*h_num_p)+in)*sizeof(particle));
140 cu_check(cuError, __FILE__, __LINE__);
141 cuError = cudaMemcpy(d_dummy_p, *d_p, (*h_num_p)*sizeof(particle), cudaMemcpyDeviceToDevice);
142 cu_check(cuError, __FILE__, __LINE__);
143 cuError = cudaFree(*d_p);
144 cu_check(cuError, __FILE__, __LINE__);
145 cuError = cudaMalloc((void **) d_p, ((*h_num_p)+in)*sizeof(particle));
146 cu_check(cuError, __FILE__, __LINE__);
147 cuError = cudaMemcpy(*d_p, d_dummy_p, (*h_num_p)*sizeof(particle), cudaMemcpyDeviceToDevice);
148 cu_check(cuError, __FILE__, __LINE__);
149 cuError = cudaFree(d_dummy_p);
150 cu_check(cuError, __FILE__, __LINE__);
151
152 // add particles
153 if (in != 0) {
154   // prepare rejection algorithm for particle velocity generation in case it's needed
155   if (vd != 0.0 && vth != 0.0) {
156     dv = (vth>fabs(vd)) ? vth/100.0 : fabs(vd)/100.0;
157     i = 0;
158     y1 = host_vdf(double(i)*dv, vth, fabs(vd));
159     do {
160       y2 = host_vdf(double(i+1)*dv, vth, fabs(vd));
161       ymax = (y1>y2) ? y1 : y2;
162       i++;
163       y1 = y2;
164     } while (ymax==y2);
165     do {
166       y2 = host_vdf(double(i+1)*dv, vth, fabs(vd));
167       i++;
168     } while (y2>0.001*ymax);
169     ymax *= 1.05;
170     xmax = double(i)*dv;
171   }
172
173   // execution configuration for pEmi kernel
174   griddim = 1;
175   blockdim = CURAND_BLOCK_DIM;
176
177   // launch kernel to add particles
178   cudaGetLastError();
179   pEmi<<<griddim, blockdim>>>(*d_p, *h_num_p, in, d_E, vth, vd, q/m, nn, L, r_p, fpt, fvt, *tin,
180                               dtin, xmax, ymax, state);
181   cu_sync_check(__FILE__, __LINE__);
182
183   // actualize time for next particle insertion
```

```
184      (*tin) += double(in)*dtin;
185
186      // actualize number of particles
187      *h_num_p += in;
188    }
189
190    return;
191  }
192
193  void calibrate_ion_flux(double *vd_i, double *dtin_i, double *d_E, double *d_phi)
194  {
195    /*-------------------------------------- function variables ---------------------------------------*/
196    // host memory
197    static const double n = init_n();
198    static const double l_p = init_l_p();
199    static const double r_p = init_r_p();
200    static const double theta = init_theta_p();
201    static const double L = init_L();
202    static const double mi = init_mi();
203    static const double kti = init_kti();
204    static const double me = init_me();
205    static const double kte = init_kte();
206    static const double vd_e = init_vd_e();
207    static const int nn = init_nn();
208
209    double phi_s, E_s, E_cs;
210    static const double increment = init_increment();
211
212    cudaError cuError;                              // cuda error variable
213
214    // device memory
215
216    /*---------------------------------------- function body -----------------------------------------*/
217
218   //---- Actualize ion drift velocity acording to the value of electric field at plasma frontier
219
220   // copy field from device to host memory
221   cuError = cudaMemcpy (&E_s, &d_E[nn-1], sizeof(double), cudaMemcpyDeviceToHost);
222   cu_check(cuError, __FILE__, __LINE__);
223
224   // evaluate theoretical field
225   phi_s = -0.5*mi*(*vd_i)*(*vd_i);
226   E_cs = (2./L)*((phi_s-2.*kti)*exp(2.*phi_s)+2.*kti*exp(3.*phi_s))/((1.+2.*phi_s-4.*kti)*exp(2.*phi_s)+
227          6.*kti*exp(3.*phi_s));
228
229   // actualize ion drift velocity
230   if (E_s<E_cs && *vd_i>-1.0/sqrt(mi)) {
231     *vd_i -= increment;
232   } else if (E_s>E_cs && *vd_i<0.0) {
233     *vd_i += increment;
234   }
235
236   // actualize sheath edge potential
237   phi_s = -0.5*mi*(*vd_i)*(*vd_i);
238   cuError = cudaMemcpy (&d_phi[nn-1], &phi_s, sizeof(double), cudaMemcpyHostToDevice);
239   cu_check(cuError, __FILE__, __LINE__);
240
241   //---- Actualize time between ion/electron insertions
242   *dtin_i = n*sqrt(kti/(2.0*PI*mi))*exp(-0.5*mi*(*vd_i)*(*vd_i)/kti);  // thermal input flux
243   *dtin_i += 0.5*n*(-(*vd_i))*(1.0+erf(sqrt(0.5*mi/kti)*(-(*vd_i))));  // drift input flux
244   *dtin_i *= exp(phi_s);                                               // density fix sheath edge
245   *dtin_i *= (r_p+L)*theta*l_p;       // number of particles that enter the simulation per unit of time
246   *dtin_i = 1.0/(*dtin_i);            // time between consecutive particles injection
247
248   return;
249  }
250
251  inline double host_vdf(double v, double vth, double vd)
252  {
253    /*-------------------------------------- function variables ---------------------------------------*/
254    // host variables definition
255
256    // device variables definition
257
258    /*---------------------------------------- function body -----------------------------------------*/
259
260    return v*exp(-(v-vd)*(v-vd)/(2.0*vth*vth));
261  }
262
263  /************************************* DEVICE KERNELS DEFINITIONS ************************************/
264
265  __global__ void pEmi(particle *g_p, int num_p, int n_in, double *g_E, double vth, double vd, double qm,
266                       int nn, double L, double r_p, double fpt, double fvt, double tin, double dtin,
267                       double xmax, double ymax, curandStatePhilox4_32_10_t *state)
268  {
269    /*--------------------------------------- kernel variables ----------------------------------------*/
270    // kernel shared memory
```

```
271   __shared__ double sh_E;
272
273   // kernel registers
274   particle reg_p;
275   int tid = (int) threadIdx.x + (int) blockIdx.x * (int) blockDim.x;
276   int tpb = (int) blockDim.x;
277   curandStatePhilox4_32_10_t local_state;
278   double2 rnd;
279   double dummy_r;
280
281   /*----------------------------------------- kernel body ------------------------------------------*/
282
283   //---- initialize shared memory
284   if (tid == 0) sh_E = g_E[nn-1];
285   __syncthreads();
286
287   //---- initialize registers
288   local_state = state[tid];
289   __syncthreads();
290
291   //---- generate particles
292   for (int i = tid; i < n_in; i+=tpb) {
293     // generate register particles position
294     reg_p.r = L;
295     // generate register particles radial velocity
296     if (vd == 0.0) {
297       rnd = curand_normal2_double(&local_state);
298       reg_p.vr = -sqrt(rnd.x*rnd.x+rnd.y*rnd.y)*vth;
299     } else if (vth != 0.0 ) {
300       do {
301         rnd = curand_uniform2_double(&local_state);
302         rnd.x *= xmax;
303         rnd.y *= ymax;
304       } while (rnd.y > device_vdf(rnd.x, vth, fabs(vd)));
305       reg_p.vr = copysignf(rnd.x, vd);
306     } else {
307       reg_p.vr = vd;
308     }
309
310     // generate register particles tangential velocity
311     rnd = curand_normal2_double(&local_state);
312     reg_p.vt = rnd.x*vth;
313
314     // simple push
315     dummy_r = reg_p.r + (fpt-(tin+double(i)*dtin))*reg_p.vr;
316     reg_p.vt *= reg_p.r/dummy_r;
317     reg_p.r = dummy_r;
318     reg_p.vr += (fvt-(tin+double(i)*dtin))*(sh_E*qm+reg_p.vt*reg_p.vt/(L+r_p));
319
320     // store new particles in global memory
321     g_p[num_p+i] = reg_p;
322   }
323   __syncthreads();
324
325   //---- store local state in global memory
326   state[tid] = local_state;
327
328   return;
329 }
330
331 __global__ void pRemover (particle *g_p, int *g_num_p, double L, int *g_num_abs_p)
332 {
333   /*--------------------------------------- kernel variables ----------------------------------------*/
334   // kernel shared memory
335   __shared__ int sh_tail;
336   __shared__ int sh_num_abs_p;
337
338   // kernel registers
339   int tid = (int) threadIdx.x;
340   int bdim = (int) blockDim.x;
341   int N = *g_num_p;
342   int ite = (N/bdim)*bdim;
343   int reg_tail;
344   particle reg_p;
345
346   /*----------------------------------------- kernel body ------------------------------------------*/
347
348   //---- initialize shared memory
349   if (tid == 0) {
350     sh_tail = 0;
351     sh_num_abs_p = 0;
352   }
353   __syncthreads();
354
355   //---- analize full batches of particles
356   for (int i = tid; i<ite; i+=bdim) {
357     // load particles from global memory to registers
```

```
358       reg_p = g_p[i];
359 
360       // analize particle
361       if (reg_p.r >= 0 && reg_p.r <= L) {
362         reg_tail = atomicAdd(&sh_tail, 1);
363       } else {
364         reg_tail = -1;
365         if (reg_p.r < 0.0) atomicAdd(&sh_num_abs_p, 1);
366       }
367       __syncthreads();
368 
369       // store accepted particles in global memory
370       if (reg_tail >= 0) g_p[reg_tail] = reg_p;
371 
372       __syncthreads();
373     }
374     __syncthreads();
375 
376     //---- analize last batch of particles
377     if (ite+tid < N) {
378       // loag particles from global memory to registers
379       reg_p = g_p[ite+tid];
380 
381       // analize particle
382       if (reg_p.r >= 0 && reg_p.r <= L) {
383         reg_tail = atomicAdd(&sh_tail, 1);
384       } else {
385         reg_tail = -1;
386         if (reg_p.r < 0.0) atomicAdd(&sh_num_abs_p, 1);
387       }
388     }
389     __syncthreads();
390 
391     // store accepted particles of last batch in global memory
392     if (ite+tid < N && reg_tail >= 0) g_p[reg_tail] = reg_p;
393 
394     // store new number of particles in global memory
395     if (tid == 0) {
396       *g_num_p = sh_tail;
397       *g_num_abs_p = sh_num_abs_p;
398     }
399 
400     return;
401   }
402 
403   __device__ inline double device_vdf(double v, double vth, double vd)
404   {
405     /*--------------------------------------- kernel variables ----------------------------------------*/
406 
407     /*----------------------------------------- kernel body ------------------------------------------*/
408 
409     return v*exp(-(v-vd)*(v-vd)/(2.0*vth*vth));
410   }
```

Code C.8: CUPIC1D2V_CP source file cc.cu

```
1   /****************************************************************************************************
2    *                                                                                                  *
3    *    This file is part of CUPIC1D2V_CP, a code that simulates the interaction between a plasma and *
4    *    a cylindrical Langmuir probe in 1D using PIC techniques accelerated with the use of GPU       *
5    *    hardware (CUDA, extension of C/C++)                                                           *
6    *                                                                                                  *
7    ****************************************************************************************************/
8 
9   #ifndef CC_H
10  #define CC_H
11 
12  /********************************************** HEADERS **********************************************/
13 
14  #include "stdh.h"
15  #include "init.h"
16  #include "random.h"
17  #include "diagnostic.h"
18  #include "cuda.h"
19  #include "dynamic_sh_mem.h"
20 
21  /***************************************** SIMBOLIC CONSTANTS ****************************************/
22 
23  #define P_RMV_BLK_SZ 1024       //block dimension for particle remover kernel
24 
25  /**************************************** FUNCTION PROTOTIPES ****************************************/
26  // host function
27  void cc(double t, int *num_i, particle **d_i, double *dtin_i, double *vd_i, double *q_pi,
28          double *d_phi, double *d_E, curandStatePhilox4_32_10_t *state);
29  void abs_emi_cc(double t, double *tin, double dtin, double kt, double m, double vd, double q,
30                  double *q_p, int *h_num_p, particle **d_p, double *d_E,
31                  curandStatePhilox4_32_10_t *state);
```

```
32 void calibrate_ion_flux(double *vd_i, double *dtin_i, double *d_E, double *d_phi);
33 inline double host_vdf(double v, double vth, double vd);
34
35 // device kernels
36 __global__ void pEmi(particle *g_p, int num_p, int n_in, double *g_E, double vth, double vd, double qm,
37                      int nn, double L, double r_p, double fpt, double fvt, double tin, double dtin,
38                      double xmax, double ymax, curandStatePhilox4_32_10_t *state);
39 __global__ void pRemover(particle *g_p, int *g_num_p, double L, int *g_num_abs_p);
40
41 // device functions
42 __device__ inline double device_vdf(double v, double vth, double vd);
43
44 #endif
```

Code C.9: `CUPIC1D2V_CP` source file `cc.h`

C.6. Diagnostic

This is the module that contains all the functions and algorithms that analyse raw data from the simulation on the fly, it also saves data into files for its subsequent analysis. (sources: `diagnostic.cu`, `diagnostic.h`)

```
1 /****************************************************************************************************
2  *                                                                                                  *
3  *    This file is part of CUPIC1D2V_CP, a code that simulates the interaction between a plasma and  *
4  *    a cylindrical Langmuir probe in 1D using PIC techniques accelerated with the use of GPU        *
5  *    hardware (CUDA, extension of C/C++)                                                            *
6  *                                                                                                  *
7  ****************************************************************************************************/
8
9   /*--------------------------------------- kernel variables ----------------------------------------*/
10  /*----------------------------------------- kernel body -------------------------------------------*/
11
12 /********************************************* HEADERS *********************************************/
13
14 #include "diagnostic.h"
15
16 /*********************************** HOST FUNCTION DEFINITIONS ***********************************/
17
18 void avg_mesh(double *d_foo, double *d_avg_foo, int *count)
19 {
20   /*-------------------------------------- function variables --------------------------------------*/
21   // host memory
22   static const int nn = init_nn();           // number of nodes
23   static const int n_save = init_n_save();   // number of iterations to average
24
25   dim3 griddim, blockdim;
26   cudaError_t cuError;
27
28   // device memory
29
30   /*---------------------------------------- function body -----------------------------------------*/
31
32   // check if restart of avg_foo is needed
33   if (*count == n_save) {
34     //reset count
35     *count = 0;
36
37     //reset avg_foo
38     cuError = cudaMemset ((void *) d_avg_foo, 0, nn*sizeof(double));
39     cu_check(cuError, __FILE__, __LINE__);
40   }
41
42   // set dimensions of grid of blocks and block of threads for kernels
43   blockdim = AVG_MESH_BLOCK_DIM;
44   griddim = int(nn/AVG_MESH_BLOCK_DIM)+1;
45
46   // call to mesh_sum kernel
47   cudaGetLastError();
48   mesh_sum<<<griddim, blockdim>>>(d_foo, d_avg_foo, nn);
49   cu_sync_check(__FILE__, __LINE__);
50
51   // actualize count
52   *count += 1;
53
54   // normalize average if reached desired number of iterations
55   if (*count == n_save ) {
56     cudaGetLastError();
57     mesh_norm<<<griddim, blockdim>>>(d_avg_foo, (double) n_save, nn);
58     cu_sync_check(__FILE__, __LINE__);
59   }
```

```
60
61   return;
62 }
63
64 void eval_df(double *d_avg_ddf, double *d_avg_vdf, double vmax, double vmin, particle *d_p, int num_p, int *count)
65 {
66   /*-------------------------------------- function variables --------------------------------------*/
67   // host memory
68   static const int n_bin_ddf = init_n_bin_ddf();    // number of bins for density distribution functions
69   static const int n_bin_vdf = init_n_bin_vdf();    // number of bins for velocity distribution functions
70   static const int n_vdf = init_n_vdf();            // number of velocity distribution functions
71   static const int n_save = init_n_save();          // number of iterations to average
72   static const double L = init_L();                 // lenght of simulation
73
74   dim3 griddim, blockdim;
75   size_t sh_mem_size;
76   cudaError_t cuError;
77
78   // device memory
79
80   /*---------------------------------------- function body -----------------------------------------*/
81
82   // check if restart of distribution functions is needed
83   if (*count == n_save) {
84     //reset count
85     *count = 0;
86
87     // reset averaged distribution functions
88     cuError = cudaMemset ((void *) d_avg_ddf, 0, n_bin_ddf*sizeof(double));
89     cu_check(cuError, __FILE__, __LINE__);
90     cuError = cudaMemset ((void *) d_avg_vdf, 0, n_bin_vdf*n_vdf*sizeof(double));
91     cu_check(cuError, __FILE__, __LINE__);
92   }
93
94   // set dimensions of grid of blocks and block of threads for kernel and shared memory size
95   blockdim = PARTICLE2DF_BLOCK_DIM;
96   griddim = int(num_p/PARTICLE2DF_BLOCK_DIM) + 1;
97   sh_mem_size = sizeof(int)*(n_bin_ddf+(n_bin_vdf+1)*n_vdf);
98
99   // call to particle2df kernel
100  cudaGetLastError();
101  particle2df<<<griddim, blockdim, sh_mem_size>>>(d_avg_ddf, n_bin_ddf, L, d_avg_vdf, n_vdf,
102                                                  n_bin_vdf, vmax, vmin, d_p, num_p);
103  cu_sync_check(__FILE__, __LINE__);
104
105  // actualize count
106  *count += 1;
107
108  // normalize average if reached desired number of iterations
109  //if (*count == n_save ) {
110    //cudaGetLastError();
111    //kernel<<<griddim, blockdim>>>();
112    //cu_sync_check(__file__, __line__);
113  //}
114
115  return;
116 }
117
118 double eval_particle_energy(double *d_phi,  particle *d_p, double m, double q, int num_p)
119 {
120   /*-------------------------------------- function variables --------------------------------------*/
121   // host memory
122   static const int nn = init_nn();        // number of nodes
123   static const double ds = init_ds();     // spacial step
124   double *h_partial_U;                    // partial energy of each block
125   double h_U = 0.0;                       // total energy of particle system
126
127   dim3 griddim, blockdim;
128   size_t sh_mem_size;
129   cudaError_t cuError;
130
131   // device memory
132   double *d_partial_U;
133
134   /*---------------------------------------- function body -----------------------------------------*/
135
136   // set execution configuration of the kernel that evaluates energy
137   blockdim = ENERGY_BLOCK_DIM;
138   griddim = int(num_p/ENERGY_BLOCK_DIM)+1;
139
140   // allocate host and device memory for block's energy
141   cuError = cudaMalloc ((void **) &d_partial_U, griddim.x*sizeof(double));
142   cu_check(cuError, __FILE__, __LINE__);
143   h_partial_U = (double *) malloc(griddim.x*sizeof(double));
144
145   // define size of shared memory for energy_kernel
146   sh_mem_size = (ENERGY_BLOCK_DIM+nn)*sizeof(double);
```

```
147
148   // launch kernel to evaluate energy of the whole system
149   cudaGetLastError();
150   energy_kernel<<<griddim, blockdim, sh_mem_size>>>(d_partial_U, d_phi, nn, ds, d_p, m, q, num_p);
151   cu_sync_check(__FILE__, __LINE__);
152
153   // copy sistem energy from device to host
154   cuError = cudaMemcpy (h_partial_U, d_partial_U, griddim.x*sizeof(double), cudaMemcpyDeviceToHost);
155   cu_check(cuError, __FILE__, __LINE__);
156
157   // reduction of block's energy
158   for (int i = 0; i<griddim.x; i++) h_U += h_partial_U[i];
159
160   //free host and device memory for block's energy
161   cuError = cudaFree(d_partial_U);
162   cu_check(cuError, __FILE__, __LINE__);
163   free(h_partial_U);
164
165   return h_U;
166 }
167
168 void particles_snapshot(particle *d_p, int num_p, string filename)
169 {
170   /*------------------------------------ function variables ---------------------------------------*/
171   // host memory
172   particle *h_p;
173   FILE *pFile;
174   cudaError_t cuError;
175
176   // device memory
177
178   /*-------------------------------------- function body -----------------------------------------*/
179
180   // allocate host memory for particle vector
181   h_p = (particle *) malloc(num_p*sizeof(particle));
182
183   // copy particle vector from device to host
184   cuError = cudaMemcpy (h_p, d_p, num_p*sizeof(particle), cudaMemcpyDeviceToHost);
185   cu_check(cuError, __FILE__, __LINE__);
186
187   // save snapshot to file
188   filename.append(".dat");
189   pFile = fopen(filename.c_str(), "w");
190   for (int i = 0; i < num_p; i++) {
191     fprintf(pFile, " %.17e %.17e %.17e \n", h_p[i].r, h_p[i].vr, h_p[i].vt);
192   }
193   fclose(pFile);
194
195   // free host memory
196   free(h_p);
197
198   return;
199 }
200
201 void save_mesh(double *d_m, string filename)
202 {
203   /*------------------------------------ function variables ---------------------------------------*/
204   // host memory
205   static const int nn = init_nn();
206   double *h_m;
207   FILE *pFile;
208   cudaError_t cuError;
209
210   // device memory
211
212   /*-------------------------------------- function body -----------------------------------------*/
213
214   // allocate host memory for mesh vector
215   h_m = (double *) malloc(nn*sizeof(double));
216
217   // copy particle vector from device to host
218   cuError = cudaMemcpy (h_m, d_m, nn*sizeof(double), cudaMemcpyDeviceToHost);
219   cu_check(cuError, __FILE__, __LINE__);
220
221   // save snapshot to file
222   filename.append(".dat");
223   pFile = fopen(filename.c_str(), "w");
224   for (int i = 0; i < nn; i++) {
225     fprintf(pFile, " %d %.17e \n", i, h_m[i]);
226   }
227   fclose(pFile);
228
229   // free host memory
230   free(h_m);
231
232   return;
233 }
```

```
234
235 void save_ddf(double *d_avg_ddf, string filename)
236 {
237   /*---------------------------------------- function variables ----------------------------------------*/
238   // host memory
239   static const double l_p = init_l_p();                 // probe lenght
240   static const double theta_p = init_theta_p();         // probe angular amplitude
241   static const double r_p = init_r_p();                 // probe radius
242   static const double L = init_L();                     // size of simulation
243   static const int n_bin_ddf = init_n_bin_ddf();        // number of bins of ddf
244   static const double bin_size = L/double(n_bin_ddf);   // size of each bin
245
246   double *h_avg_ddf;                                    // host memory for ddf
247
248   FILE *pFile;
249   cudaError_t cuError;
250
251   // device memory
252
253   /*------------------------------------------ function body ------------------------------------------*/
254
255   // allocate host memory for ddf
256   h_avg_ddf = (double *) malloc(n_bin_ddf*sizeof(double));
257
258   // copy ddf from device to host
259   cuError = cudaMemcpy (h_avg_ddf, d_avg_ddf, n_bin_ddf*sizeof(double), cudaMemcpyDeviceToHost);
260   cu_check(cuError, __FILE__, __LINE__);
261
262   // save bins to file
263   filename.append(".dat");
264   pFile = fopen(filename.c_str(), "w");
265   for (int i = 0; i < n_bin_ddf; i++) {
266     double bin_pos = (double(i)+0.5)*bin_size+r_p;
267     fprintf(pFile, " %lf %lf \n", bin_pos, h_avg_ddf[i]/(l_p*bin_size*theta_p*bin_pos));
268   }
269   fclose(pFile);
270
271   //free host memory for particle vector
272   free(h_avg_ddf);
273
274   return;
275 }
276
277 void save_vdf(double *d_avg_vdf, double vmax, double vmin, string filename)
278 {
279   /*---------------------------------------- function variables ----------------------------------------*/
280   // host memory
281   static const double r_p = init_r_p();                 // probe radius
282   static const double L = init_L();                     // size of simulation
283   static const int n_vdf = init_n_vdf();                // number of vdfs
284   static const int n_bin_vdf = init_n_bin_vdf();        // number of bins of vdf
285   static const double r_bin_size = L/double(n_vdf);     // size of spatial bins
286   const double v_bin_size = (vmax-vmin)/n_bin_vdf;      // size of velocity bins
287
288   double *h_avg_vdf;                                    // host memory for ddf
289
290   FILE *pFile;
291   cudaError_t cuError;
292
293   // device memory
294
295   /*------------------------------------------ function body ------------------------------------------*/
296
297   // allocate host memory for vdf
298   h_avg_vdf = (double *) malloc(n_vdf*n_bin_vdf*sizeof(double));
299
300   // copy vdf from device to host
301   cuError = cudaMemcpy (h_avg_vdf, d_avg_vdf, n_vdf*n_bin_vdf*sizeof(double), cudaMemcpyDeviceToHost);
302   cu_check(cuError, __FILE__, __LINE__);
303
304   // save bins to file
305   filename.append(".dat");
306   pFile = fopen(filename.c_str(), "w");
307   for (int i = 0; i < n_vdf; i++) {
308     double bin_pos = (double(i)+0.5)*r_bin_size+r_p;
309     for (int j = 0; j < n_bin_vdf; j++) {
310       double bin_vel = (double(j)+0.5)*v_bin_size+vmin;
311       fprintf(pFile, " %g %g %g \n", bin_pos, bin_vel, h_avg_vdf[j+n_bin_vdf*i]);
312     }
313     fprintf(pFile, "\n");
314   }
315   fclose(pFile);
316
317   //free host memory for particle vector
318   free(h_avg_vdf);
319
320   return;
```

```
321 }
322
323 void save_log(double t, int num_i, double U_i, double *q_pi, double vd_i, double *d_phi)
324 {
325   /*--------------------------------------- function variables ---------------------------------------*/
326   // host memory
327   double dummy_phi_p;
328   string filename = "../output/log.dat";
329   FILE *pFile;
330
331   cudaError cuError;                        // cuda error variable
332
333   // device memory
334
335   /*----------------------------------------- function body -----------------------------------------*/
336
337   // copy probe's potential from device to host memory
338   cuError = cudaMemcpy (&dummy_phi_p, &d_phi[0], sizeof(double), cudaMemcpyDeviceToHost);
339   cu_check(cuError, __FILE__, __LINE__);
340
341   // save log to file
342   pFile = fopen(filename.c_str(), "a");
343   if (pFile == NULL) {
344     printf ("Error␣opening␣log␣file␣\n");
345     exit(1);
346   } else {
347     fprintf(pFile, "␣%.17e␣%d␣%.17e␣%.17e␣%.17e␣%.17e␣\n", t, num_i, U_i, *q_pi, vd_i, dummy_phi_p);
348   }
349   fclose(pFile);
350
351   // reset negative and positive current acumulated
352   *q_pi = 0.0;
353
354   return;
355 }
356
357 /************************************ DEVICE KERNELS DEFINITIONS ************************************/
358
359 __global__ void mesh_sum(double *g_foo, double *g_avg_foo, int nn)
360 {
361   /*---------------------------------------- kernel variables ----------------------------------------*/
362   // kernel shared memory
363
364   // kernel registers
365   double reg_foo, reg_avg_foo;
366
367   int tid = (int) (threadIdx.x + blockIdx.x * blockDim.x);
368
369   /*------------------------------------------ kernel body ------------------------------------------*/
370
371   // load data from global memory to registers
372   if (tid < nn) {
373     reg_foo = g_foo[tid];
374     reg_avg_foo = g_avg_foo[tid];
375   }
376   __syncthreads();
377
378   // add foo to avg foo
379   if (tid < nn) {
380     reg_avg_foo += reg_foo;
381   }
382   __syncthreads();
383
384   // store data y global memory
385   if (tid < nn) {
386     g_avg_foo[tid] = reg_avg_foo ;
387   }
388
389   return;
390 }
391
392 __global__ void mesh_norm(double *g_avg_foo, double norm_cst, int nn)
393 {
394   /*---------------------------------------- kernel variables ----------------------------------------*/
395   // kernel shared memory
396
397   // kernel registers
398   double reg_avg_foo;
399
400   int tid = (int) (threadIdx.x + blockIdx.x * blockDim.x);
401
402   /*------------------------------------------ kernel body ------------------------------------------*/
403
404   // load data from global memory to registers
405   if (tid < nn) reg_avg_foo = g_avg_foo[tid];
406
407   // normalize avg foo
```

```
408   if (tid < nn) reg_avg_foo /= norm_cst;
409   __syncthreads();
410 
411   // store data in global memory
412   if (tid < nn) g_avg_foo[tid] = reg_avg_foo ;
413 
414   return;
415 }
416 
417 __global__ void particle2df(double *g_avg_ddf, int n_bin_ddf, double L, double *g_avg_vdf, int n_vdf,
418                             int n_bin_vdf, double vmax, double vmin, particle *g_p, int num_p)
419 {
420   /*---------------------------------------- kernel variables ----------------------------------------*/
421   // kernel shared memory
422   int *sh_ddf = (int *) sh_mem;                  // shared density distribution function
423   int *sh_vdf = &sh_ddf[n_bin_ddf];              // shared velocity distribution functions
424   int *sh_num_p_vdf = &sh_vdf[n_bin_vdf*n_vdf]; // shared number of partilces in each vdf
425 
426   // kernel registers
427   particle reg_p;
428   int bin_index;
429   int vdf_index;
430   double bin_size;
431 
432   int tidx = (int) threadIdx.x;
433   int bdim = (int) blockDim.x;
434   int tid = (int) (threadIdx.x + blockIdx.x * blockDim.x);
435 
436   /*------------------------------------------ kernel body ------------------------------------------*/
437 
438   // initialize shared memory
439   for (int i = tidx; i < n_bin_ddf+(n_bin_vdf+1)*n_vdf; i+=bdim) sh_ddf[i] = 0;
440   __syncthreads();
441 
442   // analize particles
443   if (tid < num_p) {
444     // load particle data from global memory to registers
445     reg_p = g_p[tid];
446 
447     // add information to shared density distribution functions
448     bin_size = L/n_bin_ddf;
449     bin_index = __double2int_rd(reg_p.r/bin_size);
450     atomicAdd(&sh_ddf[bin_index], 1);
451 
452     // add information to shared velocity distribution function
453     bin_size = L/n_vdf;
454     vdf_index = __double2int_rd(reg_p.r/bin_size);
455     bin_size = (vmax-vmin)/double(n_bin_vdf);
456     bin_index = __double2int_rd((reg_p.vr-vmin)/bin_size);
457     if (bin_index < 0) {
458       bin_index = 0;
459     } else if (bin_index >= n_bin_vdf) {
460       bin_index = n_bin_vdf-1;
461     }
462     atomicAdd(&sh_vdf[bin_index+vdf_index*n_bin_vdf], 1);
463     atomicAdd(&sh_num_p_vdf[vdf_index], 1);
464   }
465 
466   // syncronize threads to wait until all particles have been analized
467   __syncthreads();
468 
469   // normalize density distribution function and add it to global averaged one
470   for (int i = tidx; i < n_bin_ddf; i += bdim) {
471     atomicAdd(&g_avg_ddf[i], double(sh_ddf[i]));
472   }
473   __syncthreads();
474 
475   // normalize velocity distribution functions and add them to global averaged ones
476   for (int i = tidx; i < n_vdf*n_bin_vdf; i += bdim) {
477     if (sh_num_p_vdf[i/n_bin_vdf] != 0) {
478       atomicAdd(&g_avg_vdf[i], double(sh_vdf[i])/double(sh_num_p_vdf[i/n_bin_vdf]));
479     }
480   }
481 
482   return;
483 }
484 
485 __global__ void energy_kernel(double *g_U, double *g_phi, int nn, double ds,
486                               particle *g_p, double m, double q, int num_p)
487 {
488   /*---------------------------------------- kernel variables ----------------------------------------*/
489   // kernel shared memory
490   double *sh_phi = (double *) sh_mem;   // mesh potential
491   double *sh_U = &sh_phi[nn];           // acumulation of energy in each block
492 
493   // kernel registers
494   int tid = (int) (threadIdx.x + blockIdx.x * blockDim.x);
```

```
495   int tidx = (int) threadIdx.x;
496   int bid = (int) blockIdx.x;
497   int bdim = (int) blockDim.x;
498 
499   int ic;
500   double dist;
501 
502   particle reg_p;
503 
504   /*----------------------------------------- kernel body -------------------------------------------*/
505 
506   // load potential data from global to shared memory
507   for (int i = tidx; i < nn; i += bdim) {
508     sh_phi[i] = g_phi[i];
509   }
510 
511   // initialize energy acumulation's variables
512   sh_U[tidx] = 0.0;
513   __syncthreads();
514 
515   // analize energy of each particle
516   if (tid < num_p) {
517     // load particle in registers
518     reg_p = g_p[tid];
519     // calculate what cell the particle is in
520     ic = __double2int_rd(reg_p.r/ds);
521     // calculate distances from particle to down vertex of the cell
522     dist = fabs(__int2double_rn(ic)*ds-reg_p.r)/ds;
523     // evaluate potential energy of particle
524     sh_U[tidx] = (sh_phi[ic]*(1.0-dist)+sh_phi[ic+1]*dist)*q;
525     // evaluate kinetic energy of particle
526     sh_U[tidx] += 0.5*m*(reg_p.vr*reg_p.vr+reg_p.vt*reg_p.vt);
527   }
528   __syncthreads();
529 
530   // reduction for obtaining total energy in current block
531   for (int stride = 1; stride < bdim; stride *= 2) {
532     if ((tidx%(stride*2) == 0) && (tidx+stride < bdim)) {
533       sh_U[tidx] += sh_U[tidx+stride];
534     }
535     __syncthreads();
536   }
537 
538   // store total energy of current block in global memory
539   if (tidx == 0) g_U[bid] = sh_U[0];
540 
541   return;
542 }
```

Code C.10: CUPIC1D2V_CP source file diagnostic.cu

```
1  /****************************************************************************************************
2   *                                                                                                  *
3   *    This file is part of CUPIC1D2V_CP, a code that simulates the interaction between a plasma and  *
4   *    a cylindrical Langmuir probe in 1D using PIC techniques accelerated with the use of GPU        *
5   *    hardware (CUDA, extension of C/C++)                                                            *
6   *                                                                                                  *
7   ****************************************************************************************************/
8 
9  #ifndef DIAGNOSTIC_H
10 #define DIAGNOSTIC_H
11 
12 /********************************************** HEADERS **********************************************/
13 
14 #include "stdh.h"
15 #include "init.h"
16 #include "cuda.h"
17 
18 /*************************************** SIMBOLIC CONSTANTS ****************************************/
19 
20 #define AVG_MESH_BLOCK_DIM 512      // block dimension for mesh_sum and mesh_norm
21 #define ENERGY_BLOCK_DIM 512        // block dimension for energy solver kernel
22 #define PARTICLE2DF_BLOCK_DIM 512   // block dimension for particle2df kernel
23 
24 /*************************************** FUNCTION PROTOTIPES ***************************************/
25 // host function
26 void avg_mesh(double *d_foo, double *d_avg_foo, int *count);
27 void eval_df(double *d_avg_ddf, double *d_avg_vdf, double vmax, double vmin, particle *d_p, int num_p,
28              int *count);
29 double eval_particle_energy(double *d_phi,  particle *d_p, double m, double q, int num_p);
30 void particles_snapshot(particle *d_p, int num_p, string filename);
31 void save_mesh(double *d_m, string filename);
32 void save_ddf(double *d_avg_ddf, string filename);
33 void save_vdf(double *d_avg_vdf, double vmax, double vmin, string filename);
34 void save_log(double t, int num_i, double U_i, double *q_pi, double vd_i, double *d_phi);
35 
36 // device kernels
```

```
37 __global__ void mesh_sum(double *g_foo, double *g_avg_foo, int nn);
38 __global__ void mesh_norm(double *g_avg_foo, double norm_cst, int nn);
39 __global__ void particle2df(double *g_avg_ddf, int n_bin_ddf, double L, double *g_avg_vdf, int n_vdf,
40                             int n_bin_vdf, double vmax, double vmin, particle *g_p, int num_p);
41 __global__ void energy_kernel(double *g_U, double *g_phi, int nn, double ds,
42                               particle *g_p, double m, double q, int num_p);
43
44 // device functions
45
46 #endif
```

Code C.11: CUPIC1D2V_CP source file diagnostic.h

C.7. CUDA module

This module contains a few functions related to the use of the GPU including CUDA errors handling and intrinsic function definitions. (sources: cuda.cu, cuda.h)

```
1  /****************************************************************************************************
2   *                                                                                                  *
3   *    This file is part of CUPIC1D2V_CP, a code that simulates the interaction between a plasma and  *
4   *    a cylindrical Langmuir probe in 1D using PIC techniques accelerated with the use of GPU        *
5   *    hardware (CUDA, extension of C/C++)                                                           *
6   *                                                                                                  *
7   ****************************************************************************************************/
8  /********************************************* HEADERS **********************************************/
9
10 #include "cuda.h"
11
12 /************************************ HOST FUNCTION DEFINITIONS ************************************/
13
14 void cu_check(cudaError_t cuError, const string file, const int line)
15 {
16   /*-------------------------------------- function variables ---------------------------------------*/
17
18   /*---------------------------------------- function body -----------------------------------------*/
19
20   if (0 == cuError)
21   {
22     return;
23   } else
24   {
25     cout << "CUDA␣error␣found␣in␣file␣" << file << "␣at␣line␣" << line << ".␣(error␣code:␣" <<
26             cuError << ")" << endl;
27     cout << "Exiting␣simulation" << endl;
28     exit(1);
29   }
30 }
31
32 void cu_sync_check(const string file, const int line)
33 {
34   /*-------------------------------------- function variables ---------------------------------------*/
35   cudaError_t cuError;
36
37   /*---------------------------------------- function body -----------------------------------------*/
38
39   cudaDeviceSynchronize();
40   cuError = cudaGetLastError();
41   if (0 == cuError)
42   {
43     return;
44   } else
45   {
46     cout << "CUDA␣error␣found␣in␣file␣" << file << "␣at␣line␣" << line << ".␣(error␣code:␣" <<
47             cuError << ")" << endl;
48     cout << "Exiting␣simulation" << endl;
49     exit(1);
50   }
51 }
52
53 /*********************************** DEVICE KERNELS DEFINITIONS ***********************************/
54
55 /********************************** DEVICE FUNCTION DEFINITIONS ***********************************/
56
57 __device__ double atomicAdd(double* address, double val)
58 {
59   /*-------------------------------------- function variables ---------------------------------------*/
60   unsigned long long int* address_as_ull = (unsigned long long int*)address;
61   unsigned long long int old = *address_as_ull, assumed;
62
63   /*---------------------------------------- function body -----------------------------------------*/
64   do
```

```
65   {
66     assumed = old;
67     old = atomicCAS(address_as_ull, assumed, __double_as_longlong(val + __longlong_as_double(assumed)));
68   } while (assumed != old);
69 
70   return __longlong_as_double(old);
71 }
72 
73 __device__ double atomicSub(double* address, double val)
74 {
75   /*------------------------------------- function variables ---------------------------------------*/
76   unsigned long long int* address_as_ull = (unsigned long long int*)address;
77   unsigned long long int old = *address_as_ull, assumed;
78 
79   /*--------------------------------------- function body ------------------------------------------*/
80   do
81   {
82     assumed = old;
83     old = atomicCAS(address_as_ull, assumed, __double_as_longlong(val - __longlong_as_double(assumed)));
84   } while (assumed != old);
85 
86   return __longlong_as_double(old);
87 }
```

Code C.12: CUPIC1D2V_CP source file cuda.cu

```
1  /****************************************************************************************************
2   *                                                                                                  *
3   *    This file is part of CUPIC1D2V_CP, a code that simulates the interaction between a plasma and  *
4   *    a cylindrical Langmuir probe in 1D using PIC techniques accelerated with the use of GPU        *
5   *    hardware (CUDA, extension of C/C++)                                                           *
6   *                                                                                                  *
7   ****************************************************************************************************/
8  
9  #ifndef CUDA_H
10 #define CUDA_H
11 
12 /********************************************* HEADERS ********************************************/
13 
14 #include "stdh.h"
15 
16 /************************************** SIMBOLIC CONSTANTS ***************************************/
17 
18 /************************************** FUNCTION PROTOTIPES **************************************/
19 // host function
20 void cu_check(cudaError_t cuError, const string file, const int line);
21 void cu_sync_check(const string file, const int line);
22 
23 // device kernels
24 
25 
26 // device functions (overload atomic functions for double precision support)
27 __device__ double atomicAdd(double* address, double val);
28 __device__ double atomicSub(double* address, double val);
29 
30 #endif
```

Code C.13: CUPIC1D2V_CP source file cuda.h

C.8. Extra headers

Extra header files loaded in the previous modules. (sources: stdh.h, random.h, dynamic_sh_mem.h)

```
1  /****************************************************************************************************
2   *                                                                                                  *
3   *    This file is part of CUPIC1D2V_CP, a code that simulates the interaction between a plasma and  *
4   *    a cylindrical Langmuir probe in 1D using PIC techniques accelerated with the use of GPU        *
5   *    hardware (CUDA, extension of C/C++)                                                           *
6   *                                                                                                  *
7   ****************************************************************************************************/
8  
9  #ifndef STD_H
10 #define STD_H
11 
12 /********************************************* HEADERS ********************************************/
13 
14 #include <stdlib.h>
15 #include <math.h>
16 #include <stdio.h>
17 #include <iostream>
18 #include <fstream>
```

```
19 #include <string>
20
21 using namespace std;
22
23 /**************************************** SIMBOLIC CONSTANTS *****************************************/
24
25 #define PI 3.1415926535897932   //symbolic constant for PI
26
27 /*************************************** FUNCTION PROTOTIPES ****************************************/
28
29 struct particle
30 {
31   double r;
32   double vr;
33   double vt;
34 };
35
36 #endif
```

Code C.14: CUPIC1D2V_CP source file stdh.h

```
1  /****************************************************************************************************
2   *                                                                                                  *
3   *    This file is part of CUPIC1D2V_CP, a code that simulates the interaction between a plasma and  *
4   *    a cylindrical Langmuir probe in 1D using PIC techniques accelerated with the use of GPU        *
5   *    hardware (CUDA, extension of C/C++)                                                            *
6   *                                                                                                  *
7   ****************************************************************************************************/
8
9  #ifndef RAND_H
10 #define RAND_H
11
12 /********************************************** HEADERS *********************************************/
13
14 #include <curand_kernel.h>      //curand library for random number generation (__device__ functions)
15
16 /**************************************** SIMBOLIC CONSTANTS *****************************************/
17
18 #define CURAND_BLOCK_DIM 64      //block dimension for curand kernels
19
20 /*************************************** FUNCTION PROTOTIPES ****************************************/
21
22 #endif
```

Code C.15: CUPIC1D2V_CP source file random.h

```
1  /****************************************************************************************************
2   *                                                                                                  *
3   *    This file is part of CUPIC1D2V_CP, a code that simulates the interaction between a plasma and  *
4   *    a cylindrical Langmuir probe in 1D using PIC techniques accelerated with the use of GPU        *
5   *    hardware (CUDA, extension of C/C++)                                                            *
6   *                                                                                                  *
7   ****************************************************************************************************/
8
9  #ifndef DYNAMIC_SH_MEM_H
10 #define DYNAMIC_SH_MEM_H
11
12 // variable for allowing dynamic allocation of __shared__ memory (used in several kernels)
13 extern __shared__ float sh_mem[];
14
15 #endif
```

Code C.16: CUPIC1D2V_CP source file dynamic_sh_mem.h

C.9. Additional files

File that automates the compilation process and input file to configure simulation parameters. (sources: makefile, input_data)

```
1  # Configuration
2
3  CC = g++
4  NVCC = nvcc
5  ARCHITECTURE = sm_20
6  NVCCFLAGS = -arch=$(ARCHITECTURE) #-Xptxas -v
7  LINKERFLAGS = -arch=$(ARCHITECTURE) -lcurand
8
9  OBJECTS = main.o init.o cc.o mesh.o particles.o diagnostic.o cuda.o
10
11
```

```
12 # Makefile orders
13
14 CUPIC : $(OBJECTS)
15   $(NVCC) $(LINKERFLAGS) $(OBJECTS) -o cupic
16   rm -f *~
17   mv ./cupic ../bin/cupic
18
19 main.o : main.cu
20   $(NVCC) $(NVCCFLAGS) -dc main.cu -o main.o
21
22 init.o : init.cu init.h
23   $(NVCC) $(NVCCFLAGS) -dc init.cu -o init.o
24
25 cc.o : cc.cu cc.h
26   $(NVCC) $(NVCCFLAGS) -dc cc.cu -o cc.o
27
28 mesh.o : mesh.cu mesh.h
29   $(NVCC) $(NVCCFLAGS) -dc mesh.cu -o mesh.o
30
31 particles.o : particles.cu particles.h
32   $(NVCC) $(NVCCFLAGS) -dc particles.cu -o particles.o
33
34 diagnostic.o : diagnostic.cu diagnostic.h
35   $(NVCC) $(NVCCFLAGS) -dc diagnostic.cu -o diagnostic.o
36
37 cuda.o : cuda.cu cuda.h
38   $(NVCC) $(NVCCFLAGS) -dc cuda.cu -o cuda.o
39
40 .PHONY : clean lines
41
42 clean :
43   rm -f *.o *~
44   clear
45
46 lines :
47   git ls-files | xargs wc -l
```

Code C.17: `CUPIC1D2V_CP` compilation file `makefile`

```
1 #execution configuration
2 n_ini = 0;
3 n_prev = 0;
4 n_save = 100;
5 n_fin = 50000;
6 #plasma properties
7 ne = 1.00e15;
8 Te = 2000;
9 beta = 0.00;
10 vd_e = 0.0;
11 vd_i = -1.0e-2;
12 gamma = 7296.0;
13 #probe properties
14 radius = 3.50;
15 lenght = 2.5e-0;
16 amplitude = 5.0e-1;
17 phi_p = -25.0;
18 #sizes of simulation
19 nc = 80;
20 ds = 1.0e-1;
21 dt = 5.0e-1;
22 #diagnostic properties
23 num_of_bins_ddf = 100;
24 num_of_vdf = 100;
25 num_of_bins_vdf = 100;
26 max_v_e = 5.0;
27 min_v_e = -5.0;
28 max_v_i = 0.3;
29 min_v_i = -0.3;
30 #floating potential configuration
31 floating_potential = 0;
32 #calibration configuration
33 calibrate_ion_flux = 1;
34 increment = 1.0e-5;
35 avg_nodes = 3;
36 field_tol = -0.0e-1;
```

Code C.18: `CUPIC1D2V_CP` input file `input_data`

Bibliography

[1] H. M. Mott-Smith, "History of Plasmas", *Nature*, **volume 233**(5316):pp. 219–219 (1971), URL: http://dx.doi.org/10.1038/233219a0

[2] J. I. Fernández Palop, J. Ballesteros, M. A. Hernández, R. Morales Crespo, and S. Borrego del Pino, "A Simplified Model Joining the Sheath and the Plasma in Electronegative Plasmas", *Czechoslovak Journal of Physics*, **volume 54**(2):pp. 225–238 (2004), URL: http://dx.doi.org/10.1023/B:CJOP.0000014404.80357.d3

[3] I. Langmuir, "Positive Ion Currents in the Positive Column of the Mercury Arc", *General Electric Review*, **volume 26**(11):pp. 731–735 (1923)

[4] I. Langmuir, "Positive ion currents from the positive column of mercury arcs", *Science*, **volume 58**(1502):pp. 290–291 (1923), URL: http://dx.doi.org/10.1126/science.58.1502.290

[5] I. Langmuir, "The pressure effect and other phenomena in gaseous discharges", *Journal of the Franklin Institute*, **volume 196**(6):pp. 751–762 (1923), URL: http://dx.doi.org/10.1016/S0016-0032(23)90859-8

[6] I. Langmuir and H. M. Mott-Smith, *General Electric Review*, **volume 27**:pp. 449, 538, 616, 762, 810 (1924)

[7] H. M. Mott-Smith and I. Langmuir, "The Theory of Collectors in Gaseous Discharges", *Physical Review*, **volume 28**(4):pp. 727–763 (1926), URL: http://dx.doi.org/10.1103/PhysRev.28.727

[8] T. E. Sheridan, "Ion focusing by an expanding, two-dimensional plasma sheath", *Applied Physics Letters*, **volume 68**(14):pp. 1918–1920 (1996), URL: http://dx.doi.org/10.1063/1.115625

[9] S. Qian, H. Cao, X. Liu, and C. Ding, "Nanotube array controlled carbon plasma deposition", *Applied Physics Letters*, **volume 102**(24):p. 243109 (2013), URL: http://dx.doi.org/10.1063/1.4811747

[10] G. D. Severn, X. Wang, E. Ko, and N. Hershkowitz, "Experimental Studies of the Bohm Criterion in a Two-Ion-Species Plasma Using Laser-Induced Fluorescence", *Physical Review Letters*, **volume 90**(14):p. 145001 (2003), URL: http://dx.doi.org/10.1103/PhysRevLett.90.145001

[11] V. Demidov, C. DeJoseph, and A. Kudryavtsev, "Anomalously High Near-Wall Sheath Potential Drop in a Plasma with Nonlocal Fast Electrons", *Physical Review Letters*, **volume 95**(21):p. 215002 (2005), URL: http://dx.doi.org/10.1103/PhysRevLett.95.215002

[12] D. Lee, L. Oksuz, and N. Hershkowitz, "Exact Solution for the Generalized Bohm Criterion in a Two-Ion-Species Plasma", *Physical Review Letters*, **volume 99**(15):p. 155004 (2007), URL: http://dx.doi.org/10.1103/PhysRevLett.99.155004

[13] M. D. Campanell, A. V. Khrabrov, and I. D. Kaganovich, "Absence of Debye Sheaths due to Secondary Electron Emission", *Physical Review Letters*, **volume 108**(25):p. 255001 (2012), URL: http://dx.doi.org/10.1103/PhysRevLett.108.255001

[14] J. I. Fernández Palop, J. Ballesteros, V. Colomer, and M. A. Hernández, "Theoretical ion current to cylindrical Langmuir probes for finite ion temperature values", *Journal of Physics D: Applied Physics*, **volume 29**(11):pp. 2832–2840 (1996), URL: http://dx.doi.org/10.1088/0022-3727/29/11/017

[15] A. I. Eriksson, R. Boström, R. Gill, L. Åhlén, S. E. Jansson, J. E. Wahlund, M. André, A. Mälkki, J. A. Holtet, B. Lybekk, A. Pedersen, and L. G. Blomberg, "RPC-LAP: The Rosetta Langmuir Probe Instrument", *Space Science Reviews*, **volume 128**(1-4):pp. 729–744 (2007), URL: http://dx.doi.org/10.1007/s11214-006-9003-3

[16] N. J. T. Edberg, A. I. Eriksson, U. Auster, S. Barabash, A. Bößwetter, C. M. Carr, S. W. H. Cowley, E. Cupido, M. Fränz, K. H. Glassmeier, R. Goldstein, M. Lester, R. Lundin, R. Modolo, H. Nilsson, I. Richter, M. Samara, and J. G. Trotignon, "Simultaneous measurements of Martian plasma boundaries by Rosetta and Mars Express", *Planetary and Space Science*, **volume 57**(8-9):pp. 1085–1096 (2009), URL: http://dx.doi.org/10.1016/j.pss.2008.10.016

[17] M. Cacace, T. Batal, Y. Corre, G. Di Gironimo, J. P. Gunn, J.-Y. Pascal, and S. Salasca, "Langmuir probes design for the actively cooled divertor baffle in WEST", *Fusion Engineering and Design*, **volume 93**:pp. 15–18 (2015), URL: http://dx.doi.org/10.1016/j.fusengdes.2015.02.009

[18] B. M. Annaratone and N. S. J. Braithwaite, "A comparison of a passive (filtered) and an active (driven) probe for RF plasma diagnostics", *Measurement Science and Technology*, **volume 2**(8):pp. 795–800 (1991), URL: http://dx.doi.org/10.1088/0957-0233/2/8/014

[19] C.-S. Yip and N. Hershkowitz, "Effect of a virtual cathode on the I V trace of a planar Langmuir probe", *Journal of Physics D: Applied Physics*, **volume 48**(39):p. 395201 (2015), URL: http://dx.doi.org/10.1088/0022-3727/48/39/395201

[20] M. Zanáška, J. Adámek, M. Peterka, P. Kudrna, and M. Tichý, "Comparative measurements of plasma potential with ball-pen and Langmuir probe in low-temperature magnetized plasma", *Physics of Plasmas*, **volume 22**(3):p. 033516 (2015), URL: http://dx.doi.org/10.1063/1.4916572

[21] S. Ghosh, K. K. Barada, P. K. Chattopadhyay, J. Ghosh, and D. Bora, "Resolving an anomaly in electron temperature measurement using double and triple Langmuir probes", *Plasma Sources Science and Technology*, **volume 24**(1):p. 015017 (2015), URL: http://dx.doi.org/10.1088/0963-0252/24/1/015017

[22] J. D. Swift and M. J. R. Schwar, *Electrical Probes for Plasma Diagnostics*, Iliffe Books (1970), ISBN 9780444196941

[23] J. I. Fernández Palop, J. Ballesteros, V. Colomer, and M. Hernández, "A new smoothing method for obtaining the electron distribution function in plasmas by the numerical differentiation of the I-V probe characteristic", *Review of scientific instruments*, **volume 66**(9):pp. 4625–4636 (1995), URL: http://dx.doi.org/10.1063/1.1145300

[24] I. B. I. Bernstein and I. I. N. Rabinowitz, "Theory of Electrostatic Probes in a Low-Density Plasma", *Physics of Fluids*, **volume 2**(2):pp. 112–121 (1959), URL: http://dx.doi.org/10.1063/1.1705900

[25] J. G. Laframboise, "Theory of spherical and cylindrical langmuir probes in a collisionless, maxwellian plasma at rest", *University of Toronto Institute for Aerospace Studies*, (Report - 100):pp. 1–216 (1966). Unpublished

[26] F. W. J. Olver, D. W. Lozier, R. F. Boisvert, and C. W. Clark, *NIST Handbook of mathematical functions*, volume 5, Cambridge University Press, New York, NY (USA) (2010), ISBN 9780521140638, URL: http://dlmf.nist.gov/

[27] D. Bhom, H. E. S. Burhop, and H. S. W. Massey, *The Characteristics of Electrical Discharges in Magnetic Fields*, McGraw-Hill Book Company, Inc., New York (1949). (Eds. A. Guthrie and R. K. Walerling). The National Nuclear Energy Series, Division I: Electromagnetic Separation Project, Volume I-5

[28] S. H. Lam, "Unified Theory for the Langmuir Probe in a Collisionless Plasma", *Physics of Fluids*, **volume 8**(1):p. 73 (1965), URL: http://dx.doi.org/10.1063/1.1761103

[29] J. E. Allen, R. L. F. Boyd, and P. Reynolds, "The Collection of Positive Ions by a Probe Immersed in a Plasma", *Proceedings of the Physical Society. Section B*, **volume 70**(3):pp. 297–304 (1957), URL: http://dx.doi.org/10.1088/0370-1301/70/3/303

[30] F. F. Chen, "Numerical computations for ion probe characteristics in a collisionless plasma", *Journal of Nuclear Energy. Part C, Plasma Physics, Accelerators, Thermonuclear Research*, **volume 7**(1):pp. 47–67 (1965), URL: http://dx.doi.org/10.1088/0368-3281/7/1/306

[31] R. M. Crespo, J. I. Fernández Palop, M. A. Hernández, and J. Ballesteros, "Analytical fit of the IV probe characteristic for finite ion temperature values: Justification of the radial model applicability", *Journal of Applied Physics*, **volume 95**(6):p. 2982 (2004), URL: http://dx.doi.org/10.1063/1.1650540

[32] B. M. Annaratone, M. W. Allen, and J. E. Allen, "Ion currents to cylindrical Langmuir probes in RF plasmas", *Journal of Physics D: Applied Physics*, **volume 25**(3):pp. 417–424 (1992), URL: http://dx.doi.org/10.1088/0022-3727/25/3/012

[33] J. E. Allen, "Probe theories and applications: modern aspects", *Plasma Sources Science and Technology*, **volume 4**(2):pp. 234–241 (1995), URL: http://dx.doi.org/10.1088/0963-0252/4/2/007

[34] C. M. C. Nairn, B. M. Annaratone, and J. E. Allen, "Theory of double probes in the absence of ion saturation", *Plasma Sources Science and Technology*, **volume 4**(3):pp. 416–423 (1995), URL: http://dx.doi.org/10.1088/0963-0252/4/3/011

[35] K. U. Riemann, "The Bohm criterion and sheath formation", *Journal of Physics D: Applied Physics*, **volume 24**(4):pp. 493–518 (1991), URL: http://dx.doi.org/10.1088/0022-3727/24/4/001

[36] E. Zawaideh, F. Najmabadi, and R. W. Conn, "Generalized fluid equations for parallel transport in collisional to weakly collisional plasmas", *Physics of Fluids*, **volume 29**(2):p. 463 (1986), URL: http://dx.doi.org/10.1063/1.865731

[37] F. F. Chen, "Langmuir probes in RF plasma: surprising validity of OML theory", *Plasma Sources Science and Technology*, **volume 18**(3):p. 035012 (2009), URL: http://dx.doi.org/10.1088/0963-0252/18/3/035012

[38] M. a. Hassouba, a. R. Galaly, and U. M. Rashed, "Analysis of cylindrical Langmuir probe using experiment and different theories", *Plasma Physics Reports*, **volume 39**(3):pp. 255–262 (2013), URL: http://dx.doi.org/10.1134/S1063780X13030033

[39] E. Passoth, P. Kudrna, C. Csambal, J. F. Behnke, M. Tichý, and V. Helbig, "An experimental study of plasma density determination by a cylindrical Langmuir probe at different pressures and magnetic fields in a cylindrical magnetron discharge in heavy rare gases", *Journal of Physics D: Applied Physics*, **volume 30**(12):pp. 1763–1777 (1997), URL: http://dx.doi.org/10.1088/0022-3727/30/12/013

[40] J. M. Díaz-Cabrera, M. V. Lucena-Polonio, J. I. Fernández Palop, R. Morales Crespo, M. A. Hernández, A. Tejero-del Caz, and J. Ballesteros, "Experimental study of the ion current to a cylindrical Langmuir probe taking into account a finite ion temperature", *Journal of Applied Physics*, **volume 111**(6):p. 063303 (2012), URL: http://dx.doi.org/10.1063/1.3698313

[41] F. F. Chen, J. D. Evans, and W. Zawalski, "Calibration of Langmuir probes against microwaves and plasma oscillation probes", *Plasma Sources Science and Technology*, **volume 21**(5):p. 055002 (2012), URL: http://dx.doi.org/10.1088/0963-0252/21/5/055002

[42] C. H. Shih and E. Levi, "Determination of the collision parameters by means of Langmuir probes.", *AIAA Journal*, **volume 9**(12):pp. 2417–2421 (1971), URL: http://dx.doi.org/10.2514/3.6525

[43] J. E. Allen, "Probe theory - the orbital motion approach", *Physica Scripta*, **volume 45**(5):pp. 497–503 (1992), URL: http://dx.doi.org/10.1088/0031-8949/45/5/013

[44] I. D. Sudit and R. C. Woods, "A study of the accuracy of various Langmuir probe theories", *Journal of Applied Physics*, **volume 76**(8):p. 4488 (1994), URL: http://dx.doi.org/10.1063/1.357280

[45] V. A. Godyak, R. B. Piejak, and B. M. Alexandrovich, "Electron energy distribution function measurements and plasma parameters in inductively coupled argon plasma", *Plasma Sources Science and Technology*, **volume 11**(4):pp. 525–543 (2002), URL: http://dx.doi.org/10.1088/0963-0252/11/4/320

[46] L. S. Pilling and D. A. Carnegie, "Validating experimental and theoretical Langmuir probe analyses", *Plasma Sources Science and Technology*, **volume 16**(3):pp. 570–580 (2007), URL: http://dx.doi.org/10.1088/0963-0252/16/3/016

[47] J. M. Díaz-Cabrera, J. Ballesteros, J. I. F. Palop, and A. Tejero-del Caz, "Experimental radial motion to orbital motion transition in cylindrical Langmuir probes in low pressure plasmas", *Plasma Sources Science and Technology*, **volume 24**(2):p. 025026 (2015), URL: http://dx.doi.org/10.1088/0963-0252/24/2/025026

[48] A. A. Sonin, "Free-molecule Langmuir probe and its use in flow-field studies.", *AIAA Journal*, **volume 4**(9):pp. 1588–1596 (1966), URL: http://dx.doi.org/10.2514/3.3740

[49] J. Ballesteros, J. I. Fernández Palop, M. A. Hernández, and R. Morales Crespo, "Influence of the positive ion temperature in cold plasma diagnosis", *Applied Physics Letters*, **volume 89**(10):p. 101501 (2006), URL: http://dx.doi.org/10.1063/1.2345252

[50] R. W. Hockney and J. W. Eastwood, *Computer Simulation Using Particles*, IOP Publishing Ltd. (1988), ISBN 0-85274-392-0

[51] C. Birdsall and A. Langdon, *Plasma Physics via Computer Simulations*, Adam Hilger (1991), ISBN 0-07-005371-5

[52] D. Tskhakaya, K. Matyash, R. Schneider, and F. Taccogna, "The Particle-In-Cell Method", *Contributions to Plasma Physics*, **volume 47**(8-9):pp. 563–594 (2007), URL: http://dx.doi.org/10.1002/ctpp.200710072

[53] Nvidia Corporation, "CUDA Toolkit Documentation", URL: https://docs.nvidia.com/cuda/index.html. Accessed: 17/04/2016

[54] J. M. Díaz-Cabrera, A. Tejero-del Caz, J. I. Fernández Palop, and J. Ballesteros, "Influence of the positive ion thermal motion in the radial motion to orbital motion to cylindrical Langmuir probes in low pressure plasmas. Part I: Ar+", in *XXXII International Conference on Phenomena in Ionized Gases (ICPIG)*, Iaşi (Romanía) (2015). **P1.58**

[55] M. A. Lieberman and A. J. Lichtenberg, *Principles of Plasma Discharges and Materials Processing*, Wiley (New York) (2005), ISBN 978-0-471-72001-0

[56] A. Tejero-del Caz, J. I. Fernández Palop, J. M. Díaz-Cabrera, and J. Ballesteros, "Radial-to-orbital motion transition in cylindrical Langmuir probes studied with particle-in-cell simulations", *Plasma Sources Science and Technology*, **volume 25**(1):p. 01LT03 (2016), URL: http://dx.doi.org/10.1088/0963-0252/25/1/01LT03

[57] F. F. Chen, *Introduction to Plasma Physics*, Springer US (1974), ISBN 9978-1-4757-0459-4, URL: http://dx.doi.org/10.1007/978-1-4757-0459-4